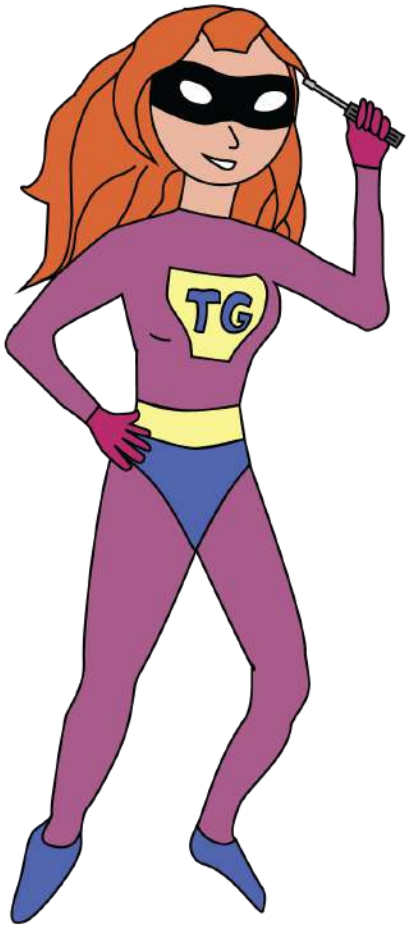
ANNA + HANNAH #ABOUTUS

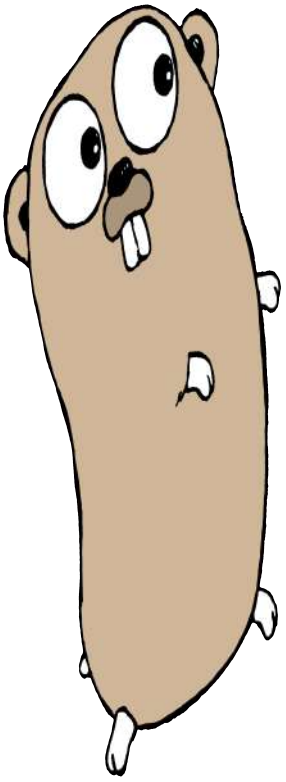


Anna Neyzberg
@Aneyzb
she/her

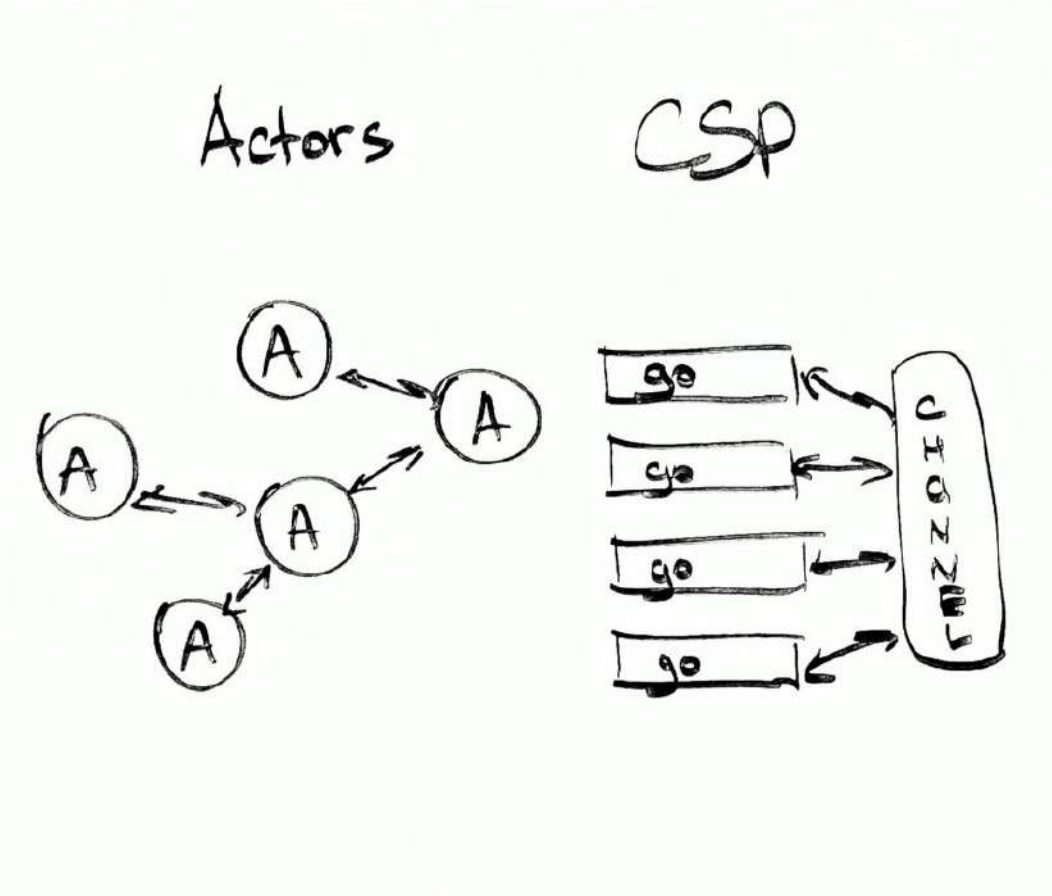
Hannah Howard
@techgirlwonder
she/her

anna@carbonfive.com
hannah@carbonfive.com

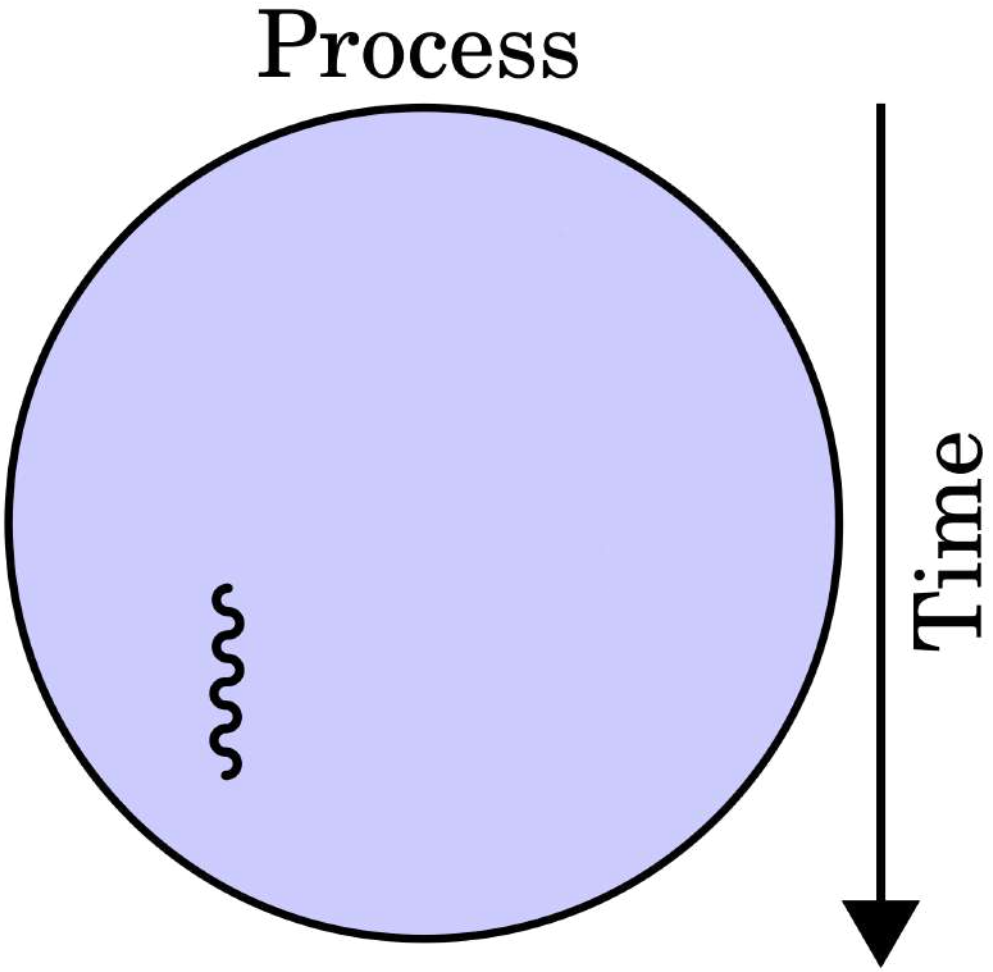
GO VS. ELIXIR

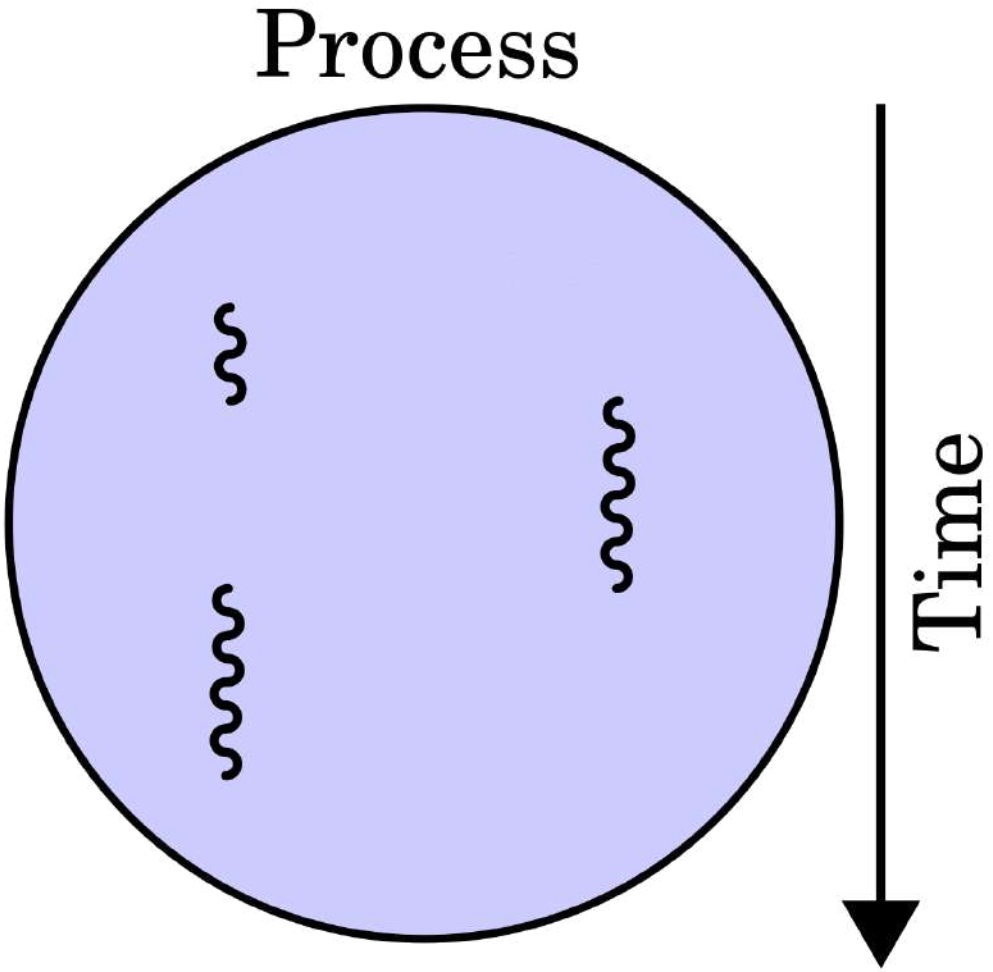


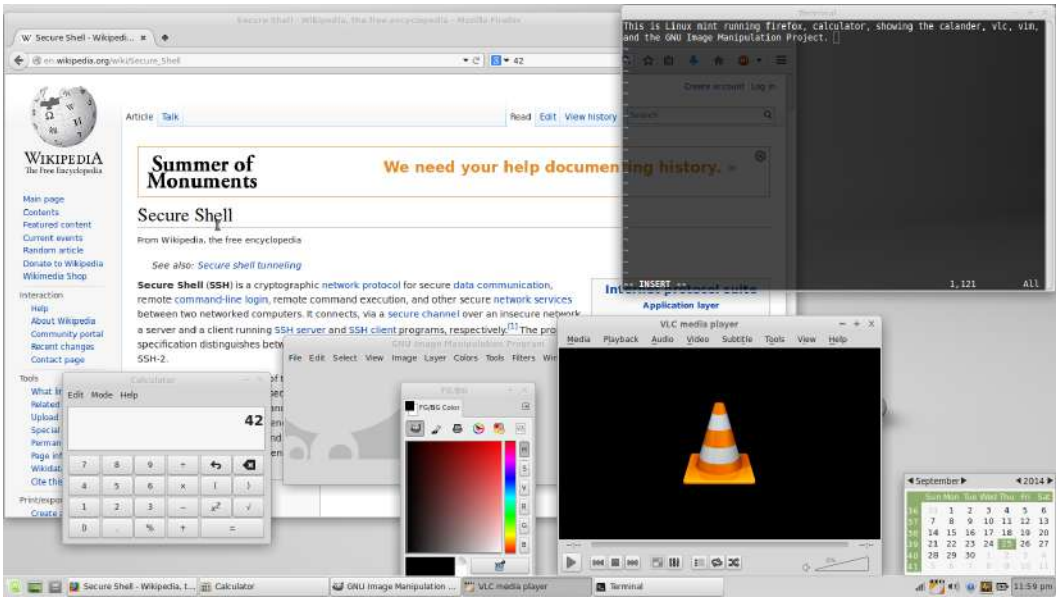
A Concurrency Comparison

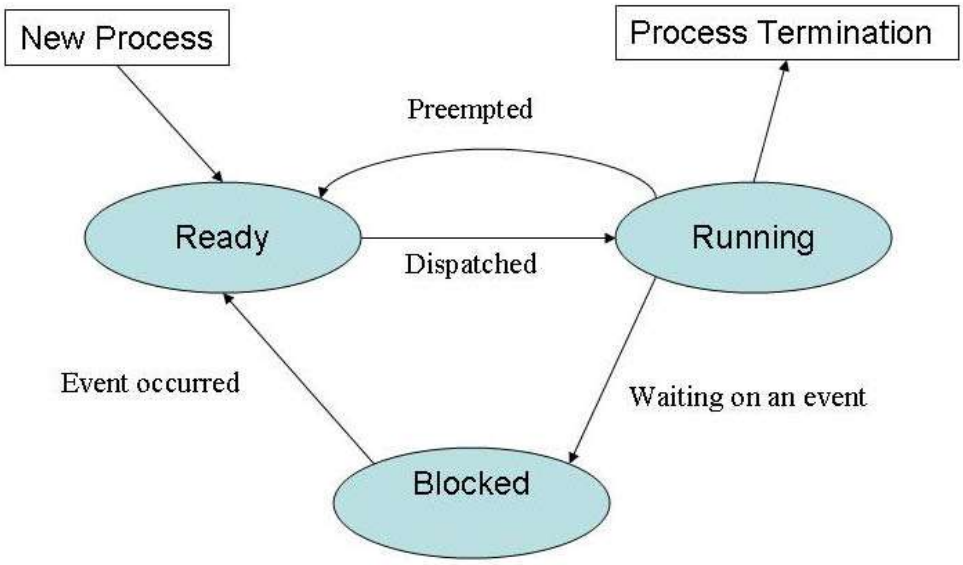

Actors
CSP
A
A
A
A
A
go
go
go
go
CHANNEL


TO START
PRESS ANY KEY
MAKE GIFS AT GIFSOUP.COM


Process
Time

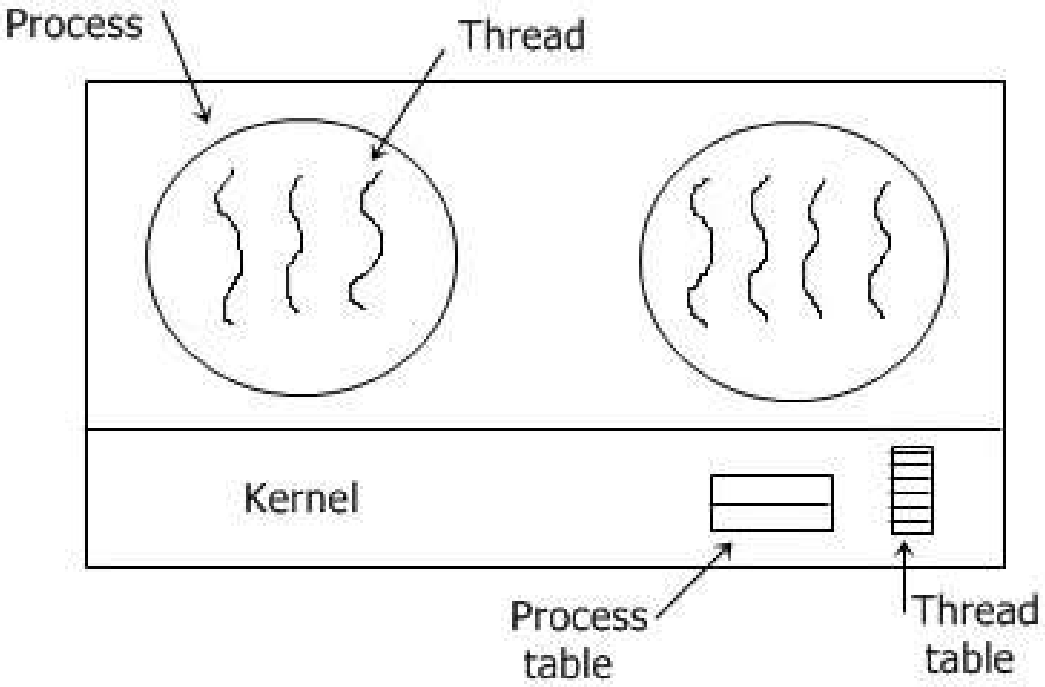

Process
Time


This is Linux mint running firefox, calculator, showing the calander, vlc, vim, and the GNU Image Manipulation Project.
Summer of Monuments
We need your help documenting history.
Secure Shell
From Wikipedia, the free encyclopedia
Secure Shell (SSH) is a cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers. It connects, via a secure channel over an insecure network, a server and a client running SSH server and SSH client programs, respectively.
VLC media player
Application layer

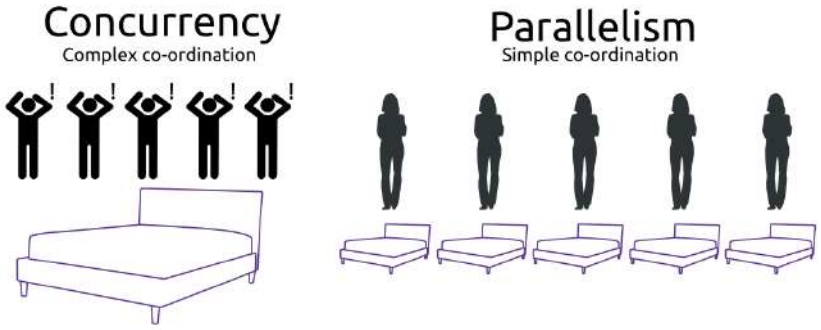

New Process
Process Termination
Preempted
Ready
Running
Dispatched
Event occurred
Waiting on an event
Blocked

BUT WE ALSO HAVE


4GIFs
.com
ABANDON THREAD


Process
Thread
Kernel
Process table
Thread table

SO HOW IS THIS RELEVANT?


Concurrency
Complex co-ordination
Parallelism
Simple co-ordination

> “Concurrency is the composition of independently executing
>
> processes, while parallelism is the simultaneous execution of computations“
>
> -

PARALLELISM IS ABOUT

EXECUTING MANY THINGS AT ONCE. IT'S FOCUS IS EXECUTION

CONCURRENCY IS ABOUT

DEALING WITH MANY THINGS AT ONCE. IT'S FOCUS IS STRUCTURE

THIS REQUIRES COORDINATION

COORDINATION INTRODUCES COMPLEXITY...

HOW DO WE

COORDINATE BETWEEN TASKS WORKING TOGETHER?

THE PROBLEM

Concurrency Without Coordination

THREADS COORDINATE BY SHARING DATA

- Use same memory space
- Use mutation on shared state to communicate
- Use various locks to prevent overwriting each others work

THREADS

Old School Concurrency

MOST WIDELY

USED PROGRAMMING LANGUAGES SUPPORT THREADS.

MO THREADS

Mo Problems

CHALLENGES WITH THREADS

1. Lots of shared mutable state
2. Dead locks / unpredictabiity
3. Expontential complexity to manage

THE ACTOR MODEL

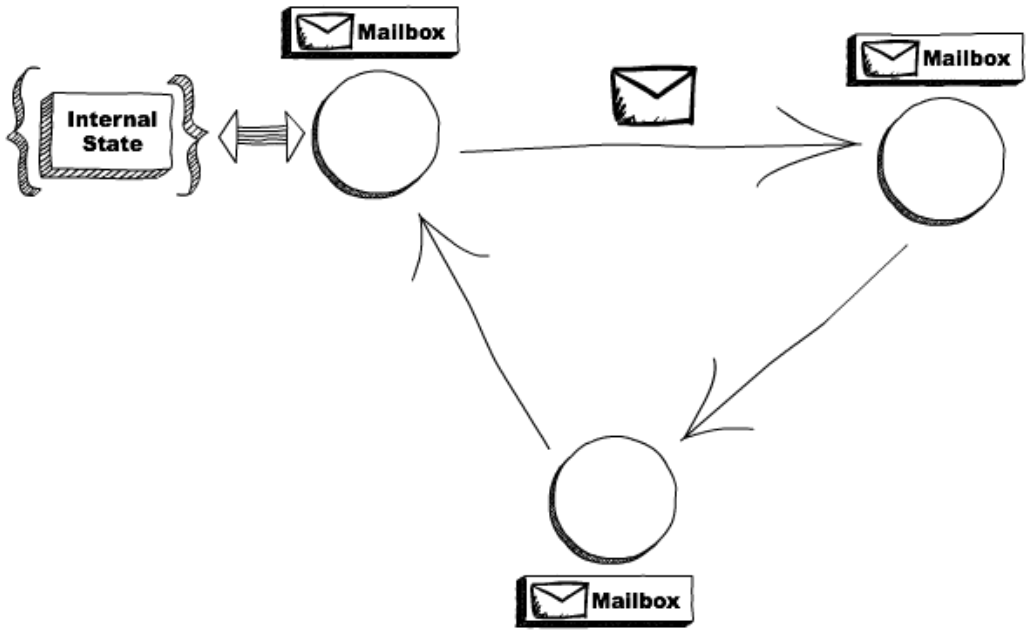
INVENTED BY CARL HEWITT 1973

ERLANG CREATED IN 1986

CONCEPTUAL MODEL TO DEAL WITH CONCURRENT COMPUTATION

DEFINES SOME GENERAL RULES FOR

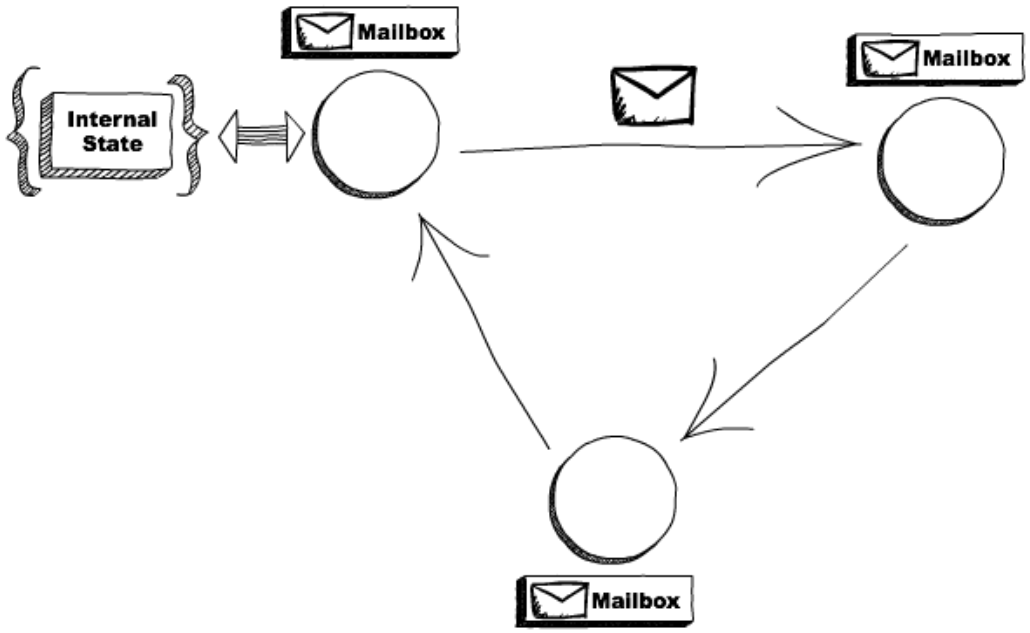
HOW THE SYSTEM'S COMPONENTS SHOULD BEHAVE AND INTERACT WITH EACH OTHER


Mailbox
Internal State
Mailbox
Mailbox

One ant is no ant.

- https://www.brianstorti.com/the-actor-model/

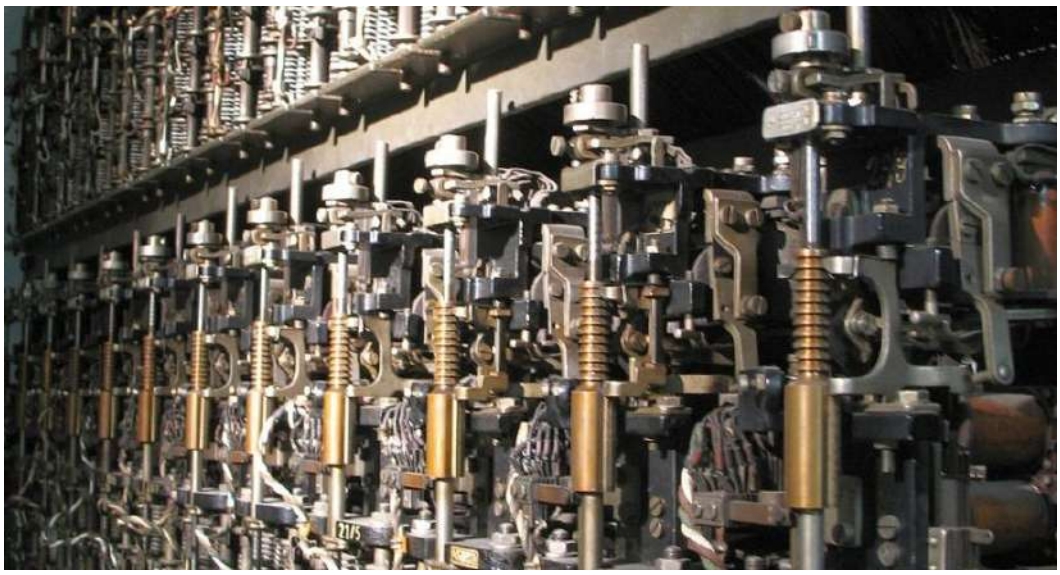
mail sending


Mailbox
Internal State
Mailbox
Mailbox

PROCCESSES

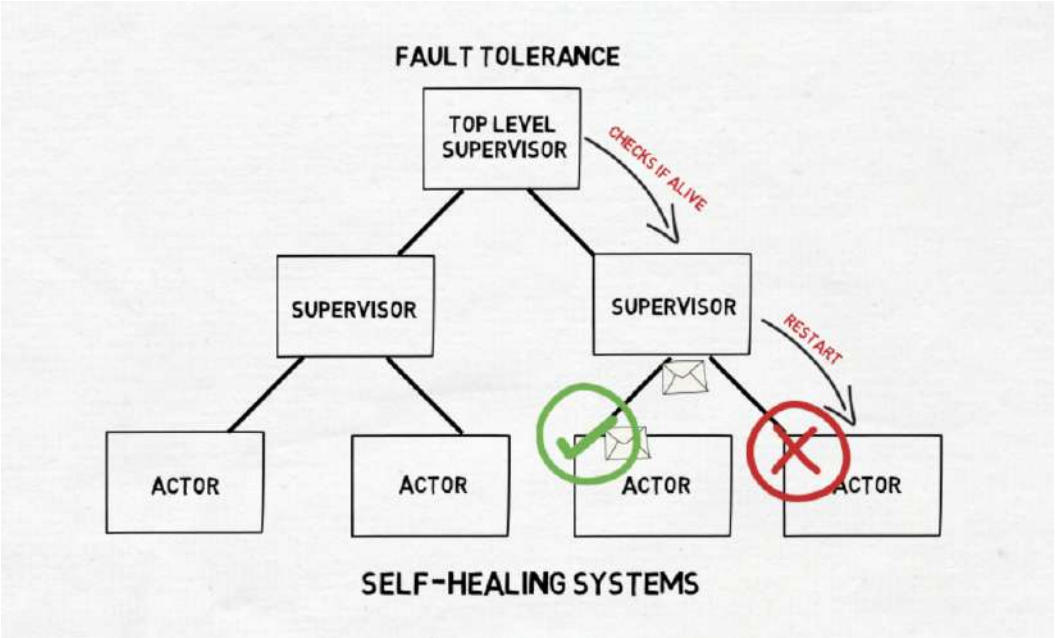
1. Are not OS processes
2. Light weight
3. Do not share memory
4. Have a unique ID

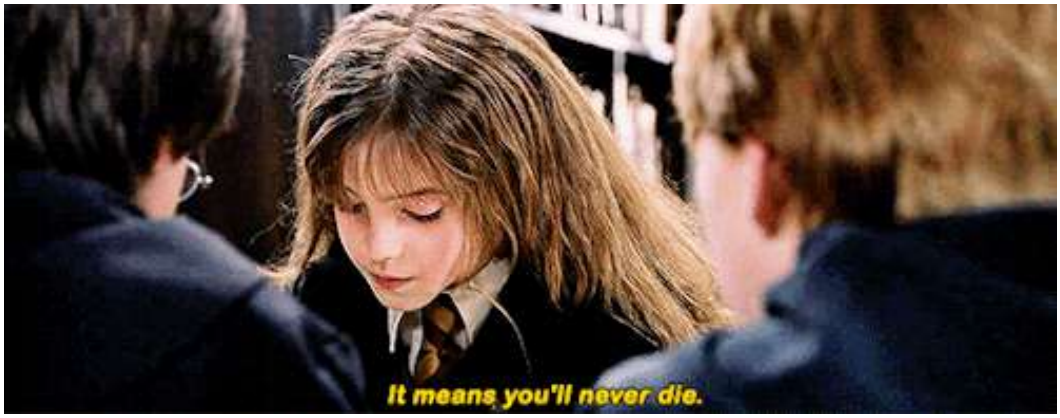
THE ACTOR MODEL IS PHYSICALLY BASED



DISTRIBUTED ELIXIR

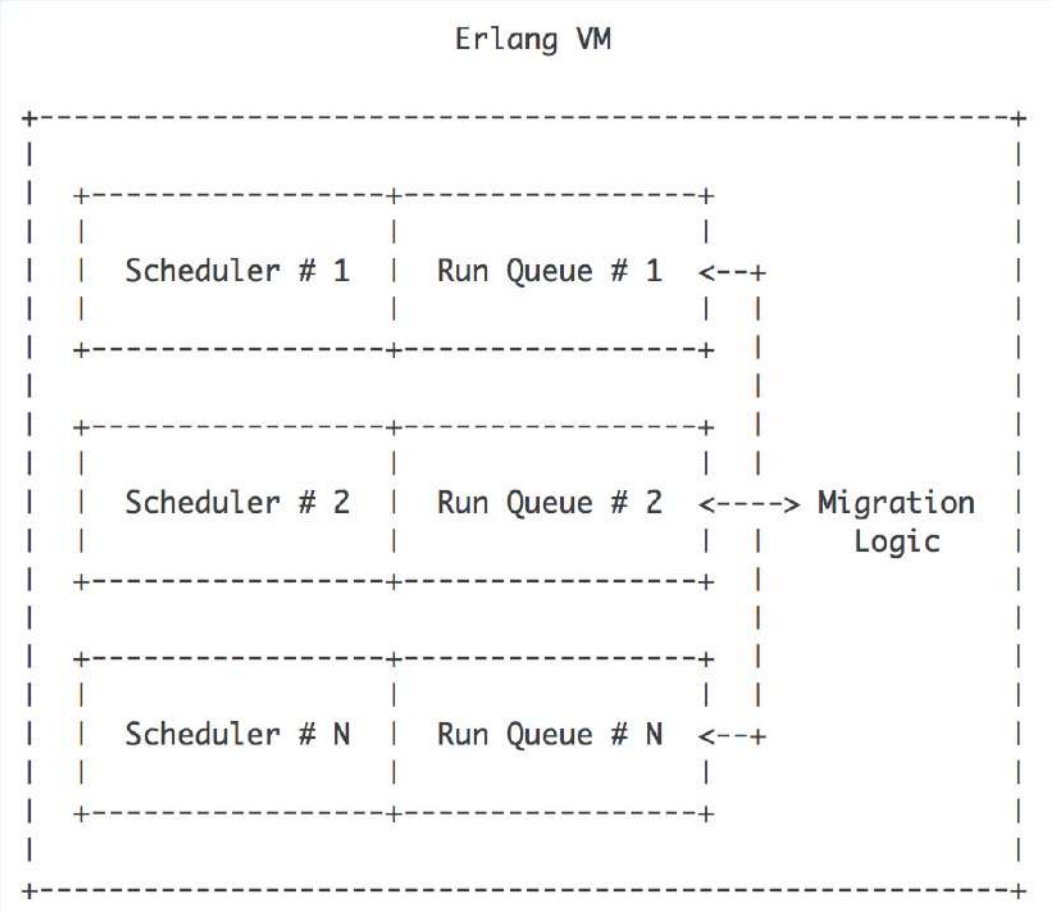
LET IT CRASH


FAULT TOLERANCE
TOP LEVEL SUPERVISOR
CHECKS IF ALIVE
SUPERVISOR
SUPERVISOR
RESTART
ACTOR
ACTOR
ACTOR
ACTOR
SELF-HEALING SYSTEMS


It means you'll never die.

THE BEAM

PREEMPTIVE SCHEDULING


Erlang VM
Scheduler # 1
Run Queue # 1
Scheduler # 2
Run Queue # 2
Migration Logic
Scheduler # N
Run Queue # N

ELIXIR PRIORITIES

1. Scalable
2. Fault Tolerant (Telco Strong)
3. Fast

GO AND CSP

CSP:

Communicating Sequential Processes

PROCESS

CSP PROCESS != OS PROCESS

CHANNELS

PROCESSES KNOW CHANNELS, NOT OTHER PROCESSES

CHOICE:

Processes choose behavior by listening on multiple channels

LOOKING AT CSP IN GO

```
package main

import "fmt"

func read(incoming <-chan string, done chan<-
struct{}) {
    readValue := <-incoming
    fmt.Printf("%s\n", readValue)
```

Wait for both to complete

```
}

func write(outgoing chan<- string, done chan<-
struct{}) {
    outgoing <- "bananas"
```

SYNCHRONOUS CHANNELS = COORDINATION

Choice in Go with select!

HOW GO IMPLEMENTS CSP

A goroutine is a lightweight thread managed by the Go runtime.

- Official Tour Of Go

GOROUTINES SHARE MEMORY

stares Britnely

BUT WAIT THERE'S MORE...

Sync Package:

ALL THE THREAD SYNC PRIMITIVES, IN GO!

> **Do not communicate by sharing memory; instead, share memory by communicating.**
>
> *- Go Docs*

> **This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.**
>
> *- Also Go Docs*

Other Go Fun Fact:
GO ROUTINES ARE SCHEDULED COOPERATIVELY!

WTF GO?

GO COMPILES TO NATIVE CODE

GO RUN-TIME IS SMALL (2MB)

GO IS (ALMOST) A SYSTEMS PROGRAMMING LANGUAGE

GO PRIORITIES

1. Ease of adoption
2. Speed
3. Flexibility

CSP VS ACTOR MODEL

CSP AND ACTOR MODEL SIMILARITIES

1. Abstract 'processes' managed by runtime
2. Share data by communicating/message passing

> **Erlang's syntax derived from**
>
> **Prolog and was heavily influenced by smalltalk, CSP and the functional programming.**
>
> *- Joe Armstrong, Erlang Creator*

Difference #1:

PROCESS IDENTITY

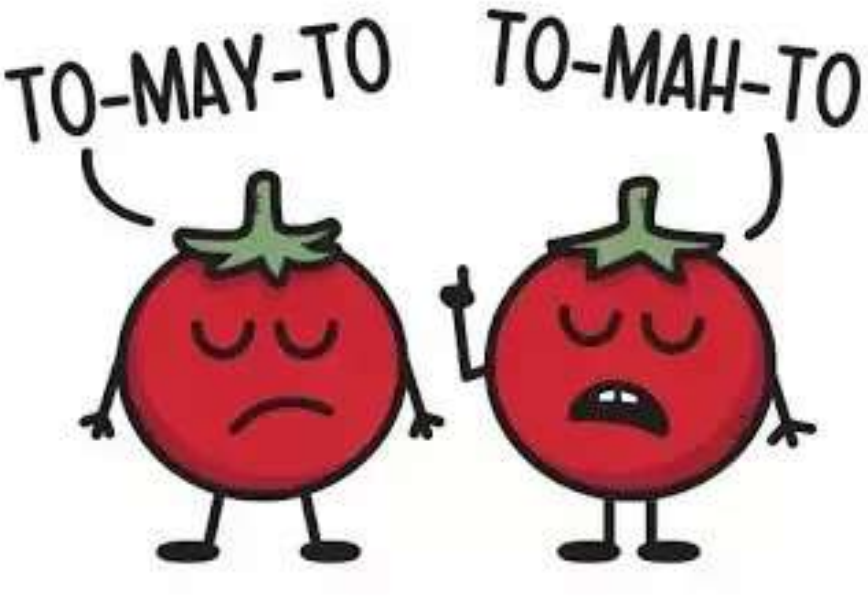
Difference #2:

DIRECT COMMUNICATION VS CHANNELS

Difference #3:

ASYNCHRONOUS VS SYNCHRONOUS MESSAGE PASSING

NO SO DIFFERENT AFTER ALL?



HOW TO GIVE CSP PROCESS AN IDENTITY

- 1 CSP channel
- + 1 CSP Process Reading From Channel
- = Channel Becomes De-facto Identity

HOW TO PASS CSP MESSAGES ASYNCHRONOUSLY

- Message Queue Process
- + Channel For Sending
- + Channel For Receiving
- = Asynchronous Message Passing

BUILDING AN UNBOUNDED CHANNEL IN GO:

https://medium.com/capital-one-tech/building-an-unbounded-channel-in-go-789e175cd2cd

A GenServer in Go!

```
1. package counter
2.
3. type counterMessage interface {
4.    handle(c * Counter)
5. }
6.
7. type Counter struct {
8.    counterMessages chan counterMessage
9.
10.    // do not touch outside run loop
11.    value int
```

HOW DO WE IMPLEMENT CHANNELS IN ELIXIR?:

```
1. defmodule ElixirConurrency.Queue do
2. def new(max_size) do
3.     { :queue.new, 0, max_size }
4.   end
5.
6.   def new(:unbuffered) do
7.     { :queue.new, 0, :unbuffered }
8.   end
9.
10.  def put(bounded_queue, item ) do
11.    case bounded_queue do
12.      { queue, len, :unbuffered } ->
if queue unbuffered
15.      { queue, len, max_size } ->
16.        if len == max_size do
17.          {:buffered, queue}
```

CONCLUSIONS

THANK YOU!

http://concurrency.techgirlwonder.com