CHICAGO

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE 2016

goto; conference

A Peek Inside

Erlang's OTP

Steve Vinoski

What is Erlang?

- Concurrency-oriented functional language
- Strong dynamic typing
- Small language, just a few elements
- Erlang VM runs BEAM bytecode
- Built-in distribution

Erlang's Origins

- Telecommunications domain, mid-80s
- Ericsson Computer Science Labs (CSL)
- Joe Armstrong, Robert Virding and Mike Williams started researching & prototyping Erlang
- Goal: develop highly reliable telephone switches better and faster

Telecom Switch Requirements

- Large number of concurrent activities
- Tolerance of software and hardware failures
- Large software systems distributed across multiple computers
- Continuous operations for years
- Live updates and maintenance

Today's Web/Cloud/µService Apps

- Large number of concurrent activities
- Tolerance of software and hardware failures
- Large software systems distributed across multiple computers
- Continuous operations for years
- Live updates and maintenance

Multi-language VM

- Erlang
- Elixir
- Lisp-Flavored Erlang (LFE) & Joxa
- Efene
- and more

Processes

Erlang Process Model

- Lightweight green threads
- One VM instance can host millions of concurrent processes
- Erlang runtime provides process scheduling and preemptive multitasking
- Processes can link to or monitor other processes

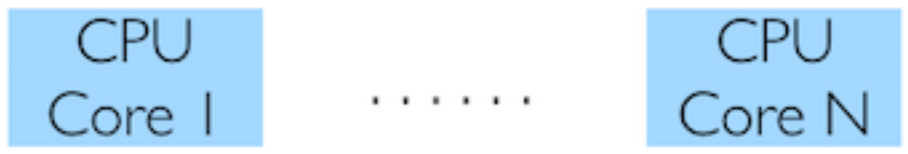
Process Execution

- A process runs a function (which may call other functions)
- The process stops when
  - its function ends
  - an unexpected exception occurs
  - something else kills it

Process Preemption

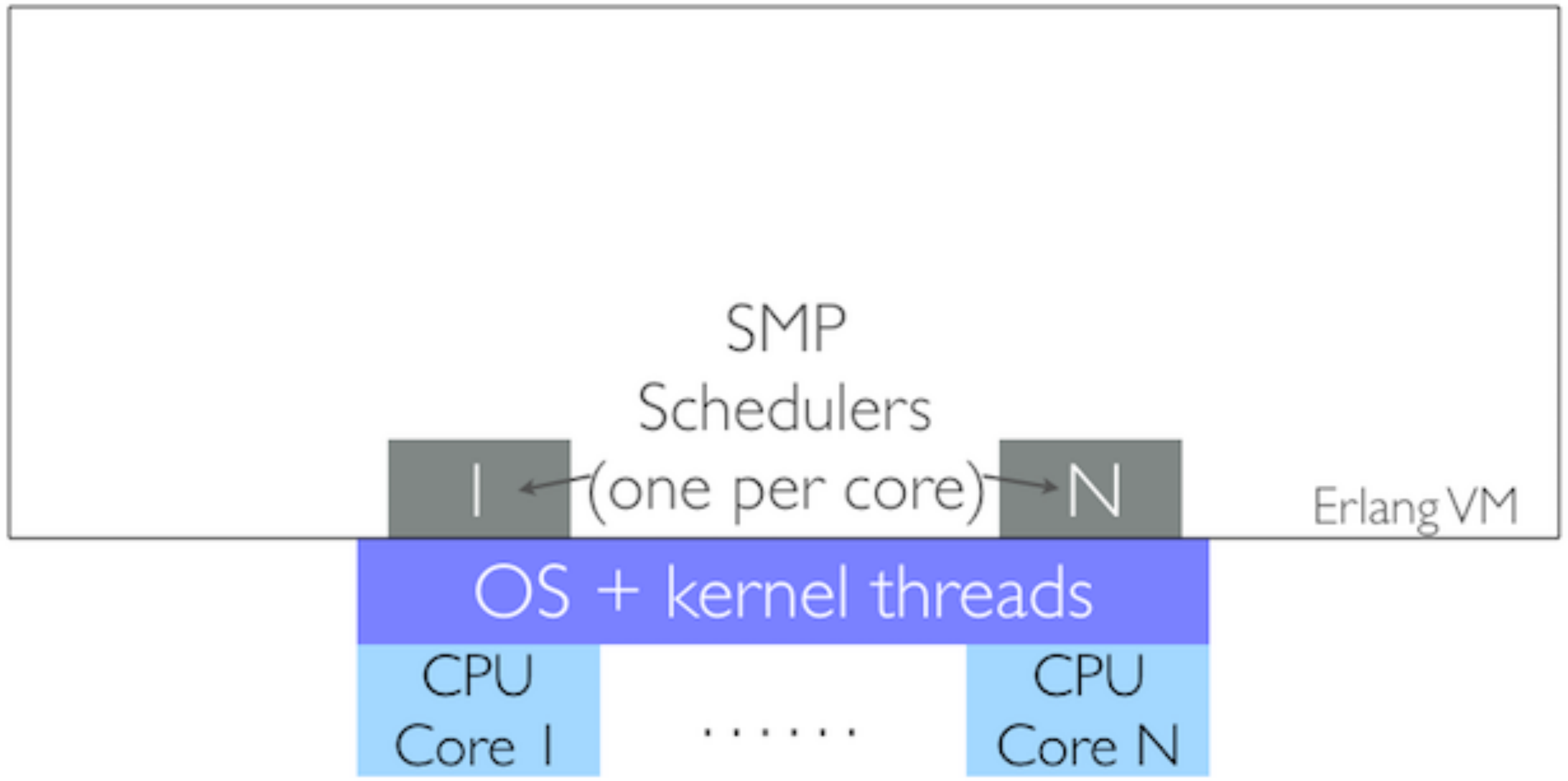
- The runtime preempts processes based on various factors:
  - executing 2000 reductions
  - waiting for a message
  - I/O
  - and more

Erlang Process Architecture

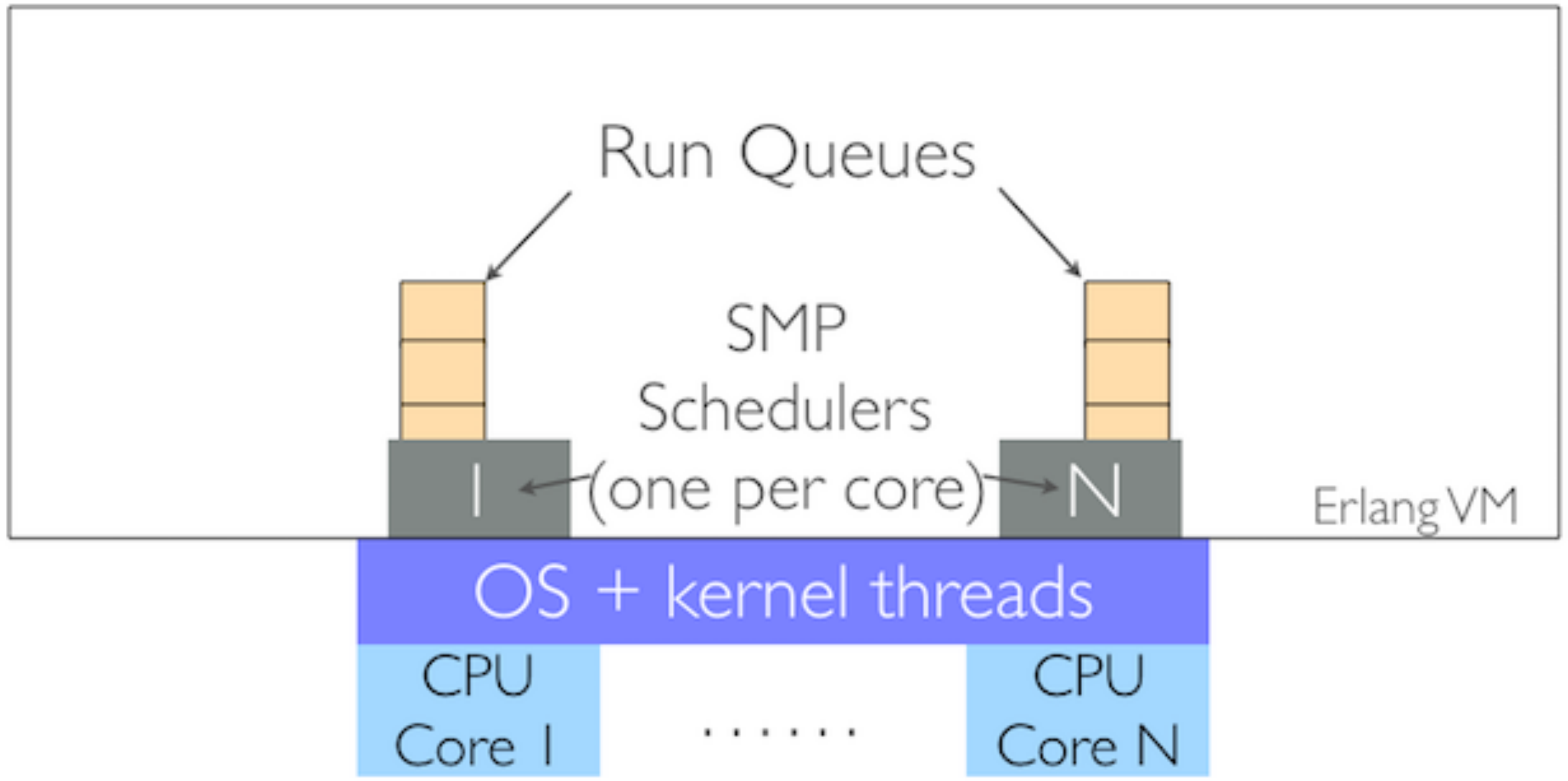


Erlang Process Architecture

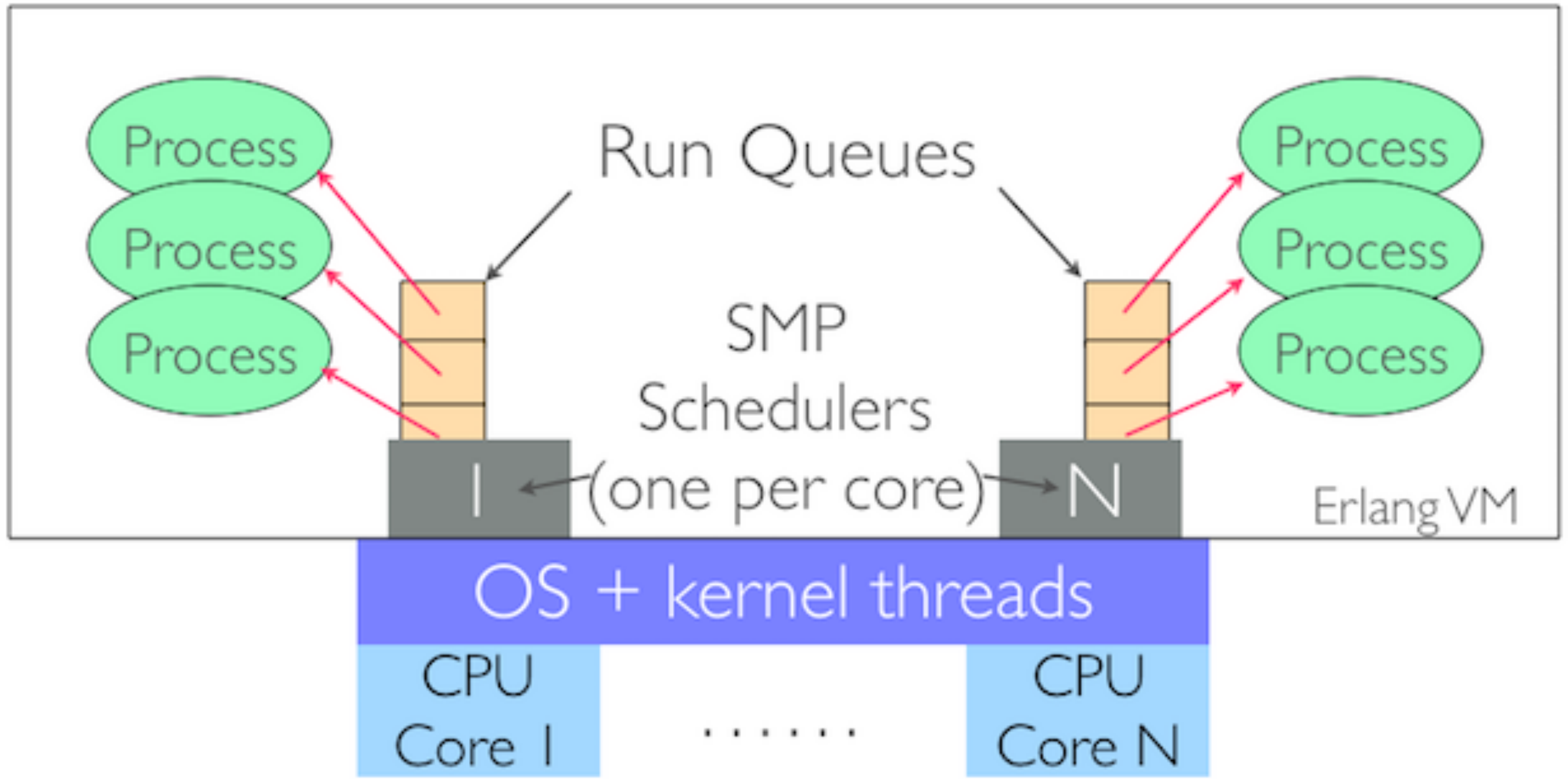
Erlang Process Architecture



Erlang Process Architecture



Erlang Process Architecture



Let It Crash

Joe Armstrong's PhD thesis recommends:

"Let some other process do the error recovery."

"If you can't do what you want to do, die."

"Let it crash."

"Do not program defensively."

Concurrency for Reliability

- Isolation: processes interact via message passing
- Recovery: via links and monitors, processes can take action when other processes die
- Distribution: process model works across nodes

Erlang Overview

- Atoms, tuples, lists, numbers, records, maps, binaries, functions, modules, process IDs, references (unique IDs), ports
- Atoms (lowercase words) are named values
- Variables (capitalized words) are immutable

- Function names and module names are atoms
- Variables live in functions
- Functions live in modules
- Functions are identified by name and arity
- Or they can be anonymous

- Functions can be
  - exported, i.e. visible to other modules
  - not exported, and so module internal
  - passed as arguments, returned from functions, stored in structures, etc.

→ Assignment is pattern matching

```erlang
%% Var is unbound, so bind to value 2 to it
Var = 2,

%% Var is bound to 2, match it against 2: success
Var = 2,

%% Var is bound to 2, match it against 3: failure
Var = 3.     % badmatch exception!
```

- For multiple clauses of same name/arity functions, matching determines which is called

```
foo([]) ->
    %% perform foo for the empty list;
foo(List) ->
    %% perform foo for the non-empty list or any other value.
```

- case expressions perform pattern matching
- case expressions are used a lot
- Pattern matching in function heads too

```
foo(A, A) ->
    %% a clause expecting two equal args

bar([H|T]) ->
    %% match the arg to a non-empty list,
    %% bind H to head, T to tail of list
```

- `ProcessId ! Message` means *send* `Message` *to the process* `ProcessId`
- Processes can have names
- Local and global registries are provided

OTP

OTP[1] Augments Erlang

- Libraries
- Tools
- Design principles

---

[1] OTP stands for "Open Telecom Platform", but it's not telecom-specific so today we just refer to it as OTP.

Design Principles

- **behaviors:** frameworks for common problems/patterns
- **supervision trees:** hierarchies of supervisor and worker processes
- **applications:** assembly of supervision trees, resources, and config data

Design Principles

- **releases:** packaged applications
- **nodes:** deployed releases
- **release handling:** upgrading/downgrading releases
- **clusters:** interconnected nodes

Other OTP Tools & Apps

- Operations, management, monitoring
- Release packaging
- Debugging, testing, performance, coverage
- And more

Behaviors

Standard Behaviors

- gen_server: supports client-server pattern
- gen_fsm and gen_statem[2]: state machines
- gen_event: event handling framework

[2] gen_statem is new in Erlang 19, June 2016

Standard Behaviors

- `supervisor`: manage worker processes
- `application`: connect your app to the rest of OTP

Purpose of Behaviors

- Separate generic reusable code from solution-specific code
- Handle generic corner cases
- Behavior modules provide generic reusable solutions to common problems

Purpose of Behaviors

- Ensure OTP compatibility so solutions can be managed properly
  - starting & stopping
  - observing & monitoring
  - debugging
  - packaging
  - live upgrades

Behavior Example

Key/Value Server Process

- Store key/value pairs
- Allow lookup by key
- Allow deletion by key
- Serve multiple client processes

Problems

- Keeping server state
- Starting and stopping
- Clients finding the server
- Handling client requests
- Dealing with errors

Process State

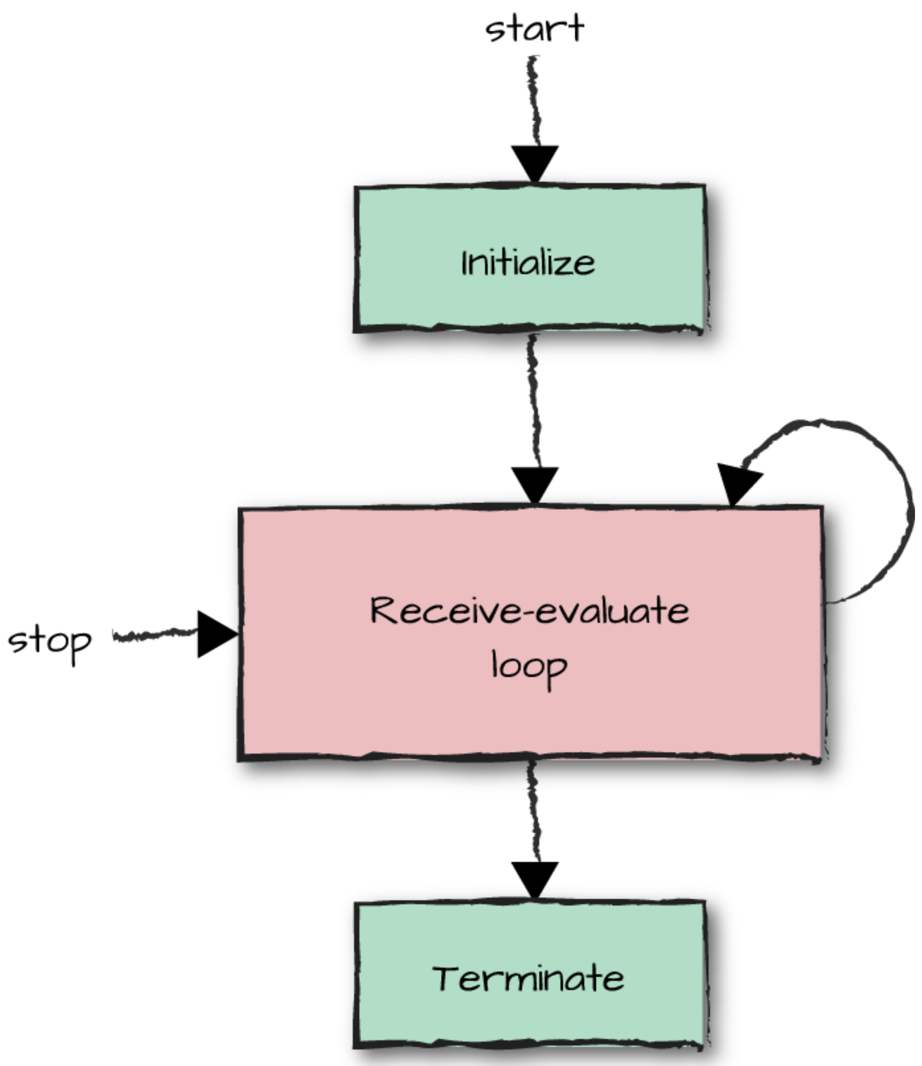
- Erlang variables are immutable
- No global variables
- How can a long-running KV server process hold state?

Receive-Evaluate Loop

- Processes execute functions
- For this case: a loop function
  - operates on current state
  - calls itself with new state
  - tail recursive , so no stack growth

```erlang
loop(State) ->
    NewState =
        receive
            %% handle messages here
            %% messages can affect State
        end,
    loop(NewState).
```

General Server Process



Server Start/Stop

- Starting: one process spawns another
- Stopping: send a stop message

Server Start

```erlang
-module(kv).
-export([start/0, stop/0]).

start() ->
    Pid = spawn(kv, loop, [#{}]),
    register(kv, Pid),
    {ok, Pid}.
```

- Spawn a process running `kv:loop/1` with initial state of `#{}` (an empty map).
- Register the process under the name `kv`

Client Code for Stop

```
stop() ->
    kv ! stop,
    ok.
```

Send a message to process kv to tell it to stop.

Server Code for Stop

```
loop(State) ->
    receive
        stop ->
            ok
    end.
```

Receive the stop atom as a message and end the recursion.

Key/Value Server API

```
-module(kv).
-export([store/2, find/1, delete/1, start/0, stop/0]).

store(Key, Value) ->
    %% store Key and Value.

find(Key) ->
    %% if Key is stored, return {Key,Value}
    %% otherwise, return false.

delete(Key) ->
    %% If Key is stored, delete it along
    %% with its value.
```

Client: Store

```
store(Key, Value) ->
    kv ! {store, Key, Value, self()},
    receive ok -> ok end.
```

- Send a `store` tuple with `Key` and `Value` to process `kv`
- Tuple contains client's process ID via `self()`
- Wait for message `ok`, then return `ok`

Server: Store

```erlang
loop(State) ->
    receive
        stop -> ok;
        {store, Key, Value, Pid} ->
            NewState = maps:put(Key, Value, State),
            Pid ! ok,
            loop(NewState)
    end.
```

- Store the key/value, creates new map
- Send `ok` back to client, then loop

Find and Delete

- Same idea: send a request tuple to the server
- Server performs the request
- Server sends response back to client

Generic vs. Specific

- What parts of this code are specific to a KV service?
- What parts are generic to client-server?

Generic Parts

- Spawning the server
- Managing loop state
- Sending client requests
- Sending server replies
- Stopping the server

Solution-Specific Parts

- Initialization at server start
- The server state
- Client request contents
- Servicing requests
- Server reply contents
- Any cleanup at server stop

Behavior Design

- Behavior generic functions implemented in a *behavior module*
- Behavior expects to be initialized with a *callback module* providing solution-specific functions
- Behavior functions call the callback module to handle everything not generic

KV using gen_server

Step 1: define kv as a gen_server callback module

```
-module(kv).
-behavior(gen_server).
```

KV using gen_server

Step 2: export API functions and callback functions

```erlang
%% API
-export([store/2, find/1, delete/1]).
-export([start_link/0, stop/0]).

%% callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

Callbacks

- `init/1` called when the `gen_server` process starts
- `handle_call/3` called to handle request/reply
- `handle_cast/2` called to handle one-way message cast

Callbacks

- `handle_info/2` called to handle any other messages
- `terminate/2` called when the process is about to stop
- `code_change/3` called during release upgrades

Starting a KV Server

```
start_link() ->
    gen_server:start_link({local, kv}, kv, [], []).
```

- Client calls `kv:start_link/0`
- That calls `gen_server:start_link/4`, with `kv` as callback module
- `gen_server` spawn_links KV process and registers it locally with the name `kv`

Starting a KV Server

```
init([]) ->
    {ok, #{}}.
```

- `gen_server` calls `kv:init/1` callback to complete solution-specific startup
- `init` returns a tuple indicating success (the atom `ok`) along with the initial process state (empty map)
- This runs in the server process

Implement store

```
store(Key, Value) ->
    gen_server:call(kv, {store, Key, Value}).
```

- `kv:store/3` calls `gen_server:call/2`
- This runs in the client process
- Note: no need to pass client pid

Implement store callback

```
handle_call({store, Key, Value}, _From, State) ->
    NewState = maps:put(Key, Value, State),
    {reply, ok, NewState};
```

- `gen_server:call/2` results in callback to `kv:handle_call/3`
- First argument is the store tuple
- Store key/value into map state, return new map as new state

Implement find callback

```
handle_call({find, Key}, _From, State) ->
    Result = case maps:find(Key, State) of
                 {ok, Value} -> {Key, Value};
                 error -> false
             end,
    {reply, Result, State}.
```

- Lookup specified `Key`
- Return `{Key,Value}` if found, `false` otherwise

Implement stop

```
stop() ->
    gen_server:stop(kv).
```

- `gen_server:stop/1` results in `terminate/2` getting called in the callback module (not shown)

`gen_server:call` Internals

Runs in the client process.

1. Monitor the `gen_server` process in case it dies or is already dead
2. Send the request to the `gen_server` process
3. Wait for reply, default 5 second timeout
4. Return reply to client

Behaviors and the `sys` Module

- Behind the scenes, behaviors handle *system messages*
- The `sys` module provides a way to work with system messages
- Handy for debugging callback modules

sys:trace/2

```
Eshell V7.3  (abort with ^G)
1> {ok, Pid} = kv:start_link().
{ok,<0.36.0>}
2> sys:trace(Pid, true).
ok
3> kv:find("GOTO").
*DBG* kv got call {find,"GOTO"} from <0.34.0>
*DBG* kv sent false to <0.34.0>, new state #{}
false
4> self().
<0.34.0>
```

sys:trace/2

```
5> kv:store("GOTO", "Chicago").
*DBG* kv got call {store,"GOTO","Chicago"} from <0.34.0>
*DBG* kv sent ok to <0.34.0>,
  new state #{[71,79,84,79]=>[67,104,105,99,97,103,111]}
ok
6> kv:find("GOTO").
*DBG* kv got call {find,"GOTO"} from <0.34.0>
*DBG* kv sent {"GOTO","Chicago"} to <0.34.0>,
  new state #{[71,79,84,79]=>[67,104,105,99,97,103,111]}
{"GOTO","Chicago"}
```

sys:get_state/1

- Examine the current loop state of a behavior:

```
7> sys:get_state(kv).
#{"GOTO" => "Chicago"}
```

- Also handy for debugging: call `sys:replace_state/2` to replace the loop state with a different state

Applications & Supervisors

`application` Behavior

- `application` provides an entry-point for an OTP app
- Allows multiple Erlang components to be combined into a release
- Apps can declare their dependencies on other apps to ensure proper start/stop order

Application Startup

- Hierarchical sequence
- The Erlang kernel starts the `application_controller` process
- `application_controller` starts an application master per app
- each application master calls app behavior `start` function
- app behavior starts the top supervisor
- top supervisor starts its child supervisors and workers

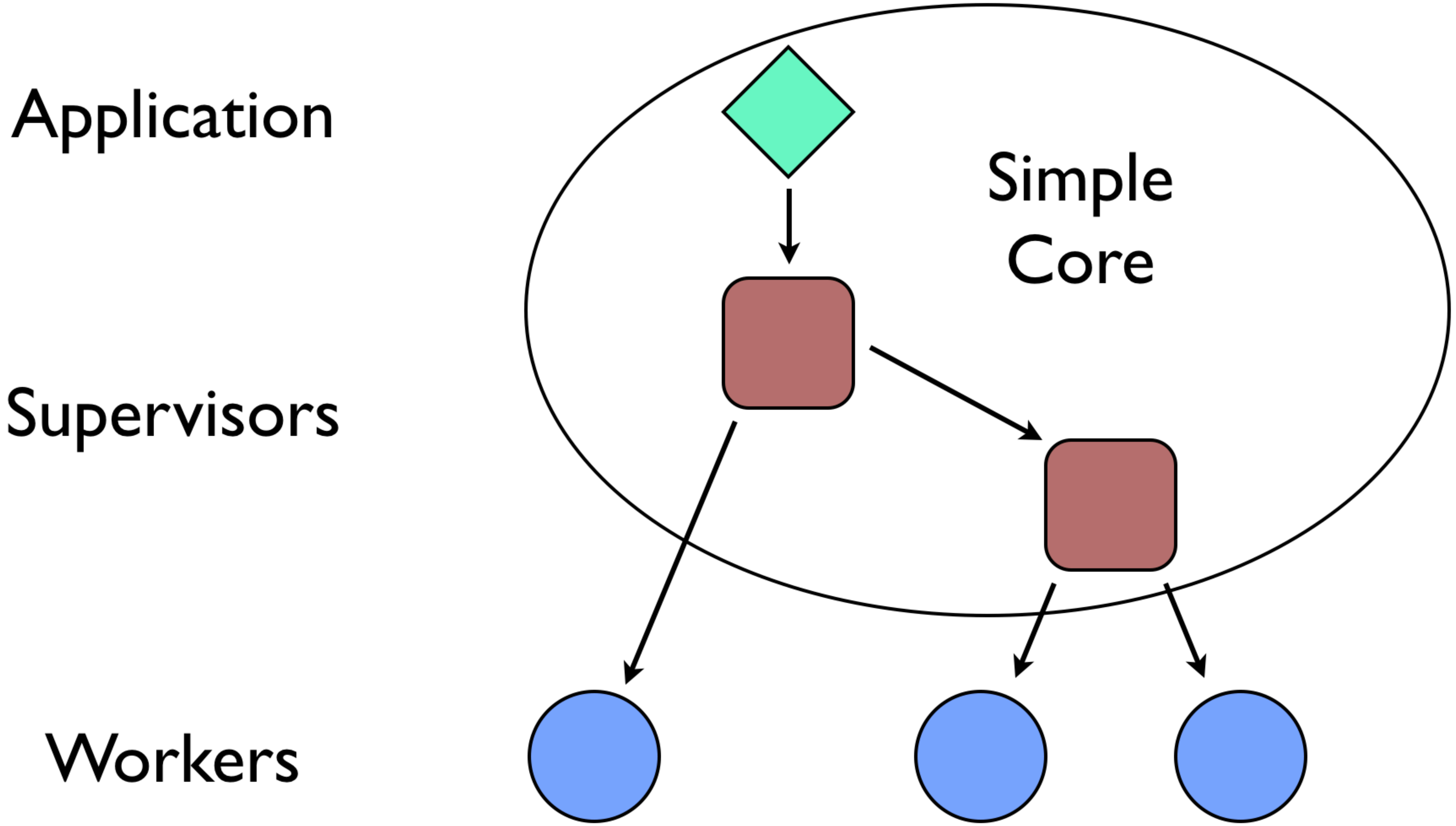
Application Example

```
-module(my_app).
-behavior(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    my_top_supervisor:start_link().

stop(_State) ->
    ok.
```

Application modules are rarely more complicated than this.

Workers and Supervisors

- Workers implement application logic
- Supervisors start child workers or child supervisors
- Linked to child processes
- Take action when a child process dies



The simple core provides a stable base for the entire application

Supervisor Features

- Restart strategies
  - `one_for_one`: a crashed child is restarted
  - `one_for_all`: a child crashes, all are restarted
  - `rest_for_one`: crashed child and those after it are restarted
  - `simple_one_for_one`: used for children added dynamically
- Max number of restarts per time period
  - supervisor dies if exceeded
  - prevents getting stuck in crash-restart loops

Supervisor Features

- Child specifications tell the supervisor how to start each child
- For example, for `kv`:

```
#{id => kv,
  start => {kv, start_link, []},
  restart => permanent,
  shutdown => 2000,
  type => worker,
  modules => [kv]}.
```

Process Problems

In the original solution:

- What if the server dies?
- What if the server dies while a client is waiting?
- What if the server takes too long to process a request?

Process Problems Solved

In the `gen_server` solution:

- If the server dies, supervisor restarts it
- If the server dies while a client waits, client's process monitor detects it, client exits with an error
- If the server takes too long, client exits with `timeout` (default 5 seconds)
- The standard behaviors handle all sorts of corner cases that are easy to miss

Summary

Benefits of Behaviors

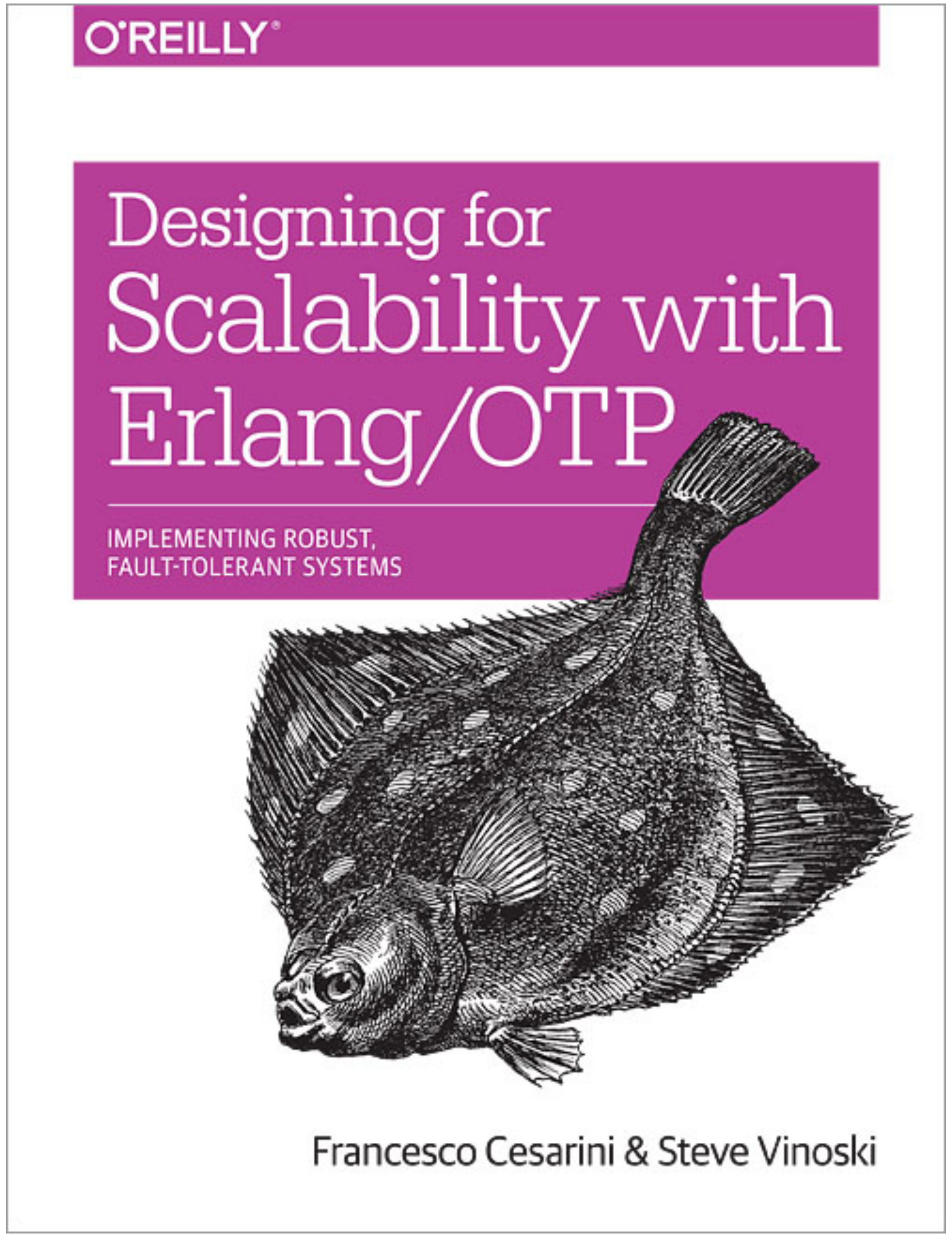
- Handling tricky corner cases
- Standardized frameworks provide reusable solutions, common vocabulary
- Used in all non-trivial Erlang-based systems
- Erlang developers understand them and can easily read them
- Features honed and proven across countless projects, many years in production

Much More to Explore

- Other behaviors
- Writing your own behaviors
- Packaging, deploying
- Live upgrades
- Monitoring, tracing, logging

For More Information

- *Designing for Scalability with Erlang/OTP*, Cesarini & Vinoski
- *Erlang Programming*, Cesarini & Thompson
- *Stuff Goes Bad: Erlang in Anger*, Hebert (https://www.erlang-in-anger.com)
- For Elixir see http://elixir-lang.org/




goto;
conference
Let us know what you think
Please
Remember to rate this session
Thank you!
follow us @gotochgo