WEEK-4 PRESENTED BY NOLLEH

LEARN U ERLANG

CHAPTER. 9

A SHORT VISIT TO DATA STRUCTURES

WON'T BE TOO LONG, PROMISED!

- now is the time, you will worry about …
  - can I build 'real useful' programs ?
- to do that, you need to know
  - more data types, provided to programmers in the erlang standard library.

RECORDS

- useful when handle small data structure where you want to access the attributes by name directly
- declare

```
-module(records).
-compile(export_all).

-record(robot, {name,
                type=industrial,
                hobbies,
                details=[]}).
```

RECORDS

- using it

```
first_robot() ->
  #robot{name="Mechatron",
         type=handmade,
         details=["Moved by a small man inside"]}.
```

RECORDS

- and running it,
  - uh … hardly distinct it ‘AS RECORD’…

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,
       ["Moved by a small man inside"]}
```

RECORDS

- there is a better way.

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,
       hobbies = undefined,
       details = ["Moved by a small man inside"]}
```

- erlang’s default value is for you..

RECORDS

- rr() can also take wild card, with list as second argument to specify which records to load

```
rr ('*', [Record1, Record2, ... ])
```

- shells command to define record

```
rd(Name, Definition)
```

RECORDS

- shells command to unload record

```
rf(),  rf(Name),  rf([Names])
```

- shells command to listing records

```
rl(), rl(Name), rl([Names])
```

- shells command to tuple to a record

```
rp(Term)
```

RECORDS

- extract its value with dot syntax

```
5> Crusher = #robot{name="Crusher", hobbies=[“Crushing
                                people","petting cats"]}.
    #robot{name = “Crusher", type = industrial,
    hobbies = ["Crushing people","petting cats"],
    details = []}
6> Crusher#robot.hobbies.
    ["Crushing people","petting cats"]
```

- oh.. #robot part is no way to avoid..

RECORDS

- worse case

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
  #robot{name = undefined,type = industrial,
  hobbies = undefined,
  details = #robot{name = "erNest",type = industrial,
                   hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

- parantheses is mandatory. (version < R14A)

RECORDS

- can be used in pattern match, guard…

```
-record(user, {id, name, group, age}).
admin_panel(#user{name=Name, group=admin}) ->
  Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
  Name ++ " is not allowed".

adult_section(U = #user{}) when U#user.age >= 18 ->
  allowed;
adult_section(_) ->
  forbidden.
```

RECORDS

- using it

```
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin,
age=96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users,
 age=66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=users,
age=72}).
allowed
```

RECORDS

- record can be pattern matched with only interesting parts.
  - when it tuple,.. u must type all...

```
function({record, _, _, ICareAboutThis, _, _}) -> ....
```

RECORDS

- update record (compiler tricky setelement/3)

```
repairman(Rob) ->
  Details = Rob#robot.details,
  NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
  {repaired, NewRob}.
```

```
16> c(records).
  {ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying to
 have feelings"]}).
  {repaired,#robot{name = "Ulbert",type = industrial,
  hobbies = ["trying to have feelings"],
  details = ["Repaired by repairman"]}}
```

RECORDS

- sharing record for modules

```
%% this is a .hrl (header) file.
-record(included, {some_field,
  some_default = "yeah!",
  unimaginative_name}).
```

```
-include("records.hrl").
included() -> #included{some_field="Some value"}.
```

RECORDS

- using it

```
18> c(records).
  {ok,records}
19> rr(records).
  [included,robot,user]
20> records:included().
  #included{some_field = "Some value",some_default = "yeah!",
    unimaginative_name = undefined}
```

- but keep record definition in local as possible as you can
  - name clash / readability / maintability..

KEY-VALUE STORES

- show list of documents...

- proplist

```
[{Key,Value}]
```

- no other rules

```
proplists:delete/2, proplists:get_value/2, proplists:get_all_values/2,
proplists:lookup/2, proplists:lookup_all/2
```

KEY-VALUE STORES

- proplist
  - no func to update its' elements

```
[NewElement|OldList],   lists:keyreplace/4
```

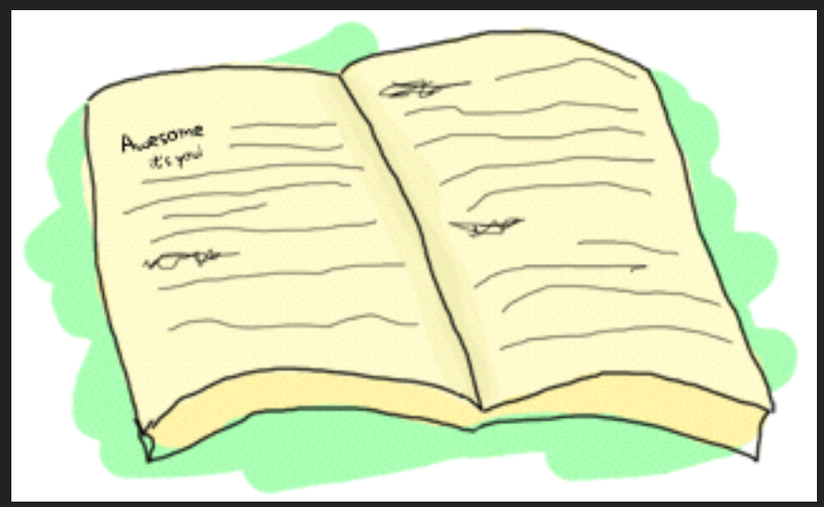
- They're more of a common pattern that appears when using lists and tuples to represent some object or item; the proplists module is a bit of a toolbox over such a pattern.

KEY-VALUE STORES

- orddict
  - unique key / sorted

orddict:store/3, orddict:find/2, orddict:fetch/2, orddict:erase/2

- efficient up to about 75 elements.



KEY-VALUE STORES

- dict
  - scale up from orddict

dict:store/3, dict:find/2, dict:fetch/2, dict:erase/2

KEY-VALUE STORES

- gb_trees
  - general balanced trees
    - 2 mode
      - smart mode (where you know your structure in out)
      - naive mode (can’t assume much about it)

KEY-VALUE STORES

- naive mode function

```
gb_trees:enter/3, gb_trees:lookup/2, gb_trees:delete_any/2
```

- smart mode function

```
gb_trees:insert/3, gb_trees:get/2, gb_trees:update/3,
gb_trees:delete/2, gb_trees:map/2
```

KEY-VALUE STORES

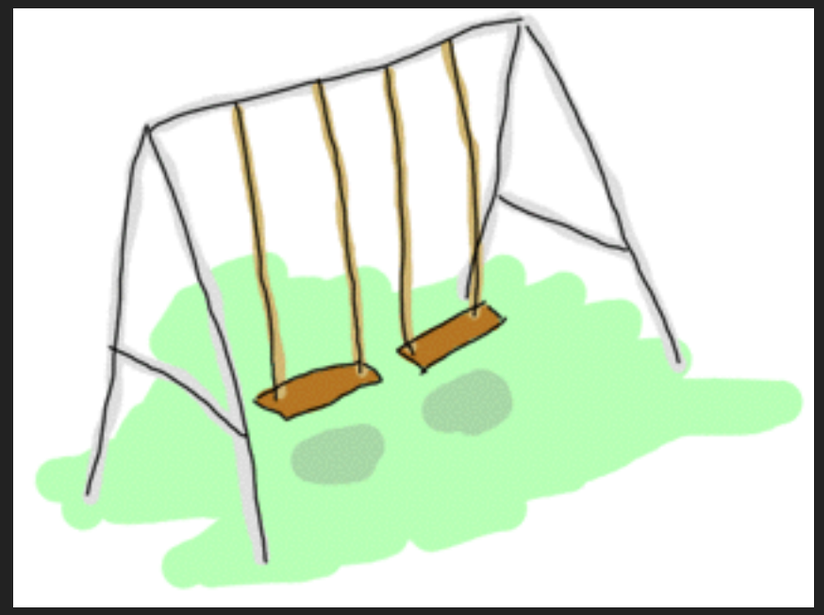
- gb_tree need to balance itself when the tree is updated.
  - 'smart' function all assume that the key is present in the tree: this lets you skip all the safety checks and results in faster times.
- dict has little faster when read, gb_tree is for others.
- dict has fold operation
- gbtree has iterator. gb_trees:next(Iterator)
  - your own recursive function
- gbtree has fast accessibility to gb_trees:smallest/1 and gb_trees:largest/1

ARRAYS

- key is simply numbers
- array : http://erldocs.com/18.0/stdlib/array.html
- not able to have such things as constant-time insertion or lookup
- rarely used in practice
- instead, Ports , C-Nodes, Linked in drivers and NIFs (Experimental, R13B03+).
- indexing with regular expression..

A SET OF SETS

- sets are groups of unique elements that you can compare and operate on: find which elements are in two groups, in none of them, only in one or the other, etc.
- 4 sets for set (no best impletation)
  - ordsets, sets, gb_sets and sofs



A SET OF SETS

- ordsets
  - implemented as sorted list
  - useful for small set, slowest set
- simplest, more readable

```
ordsets:new/0, ordsets:is_element/2, ordsets:add_element/2,
ordsets:del_element/2, ordsets:union/1, ordsets:intersection/1,
```

A SET OF SETS

- sets
  - top of structure is similar with one of used in dict
  - same interface with ordsets - scale much better
  - Like dictionaries, they're especially good for read-intensive manipulations, like checking whether some element is part of the set or not.

A SET OF SETS

- gb_sets
  - faster when considering operations different than reading, leaving you with more control
- smart vs. naive functions
- iterators
- quick access to the smallest and largest values

A SET OF SETS

- sofs
  - sorted lists, stuck inside a tuple with some metadata
  - choose when you want to have full control over relationships between sets, families, enforce set types
  - if you need mathematics concept rather than 'just' groups of unique elements

A SET OF SETS

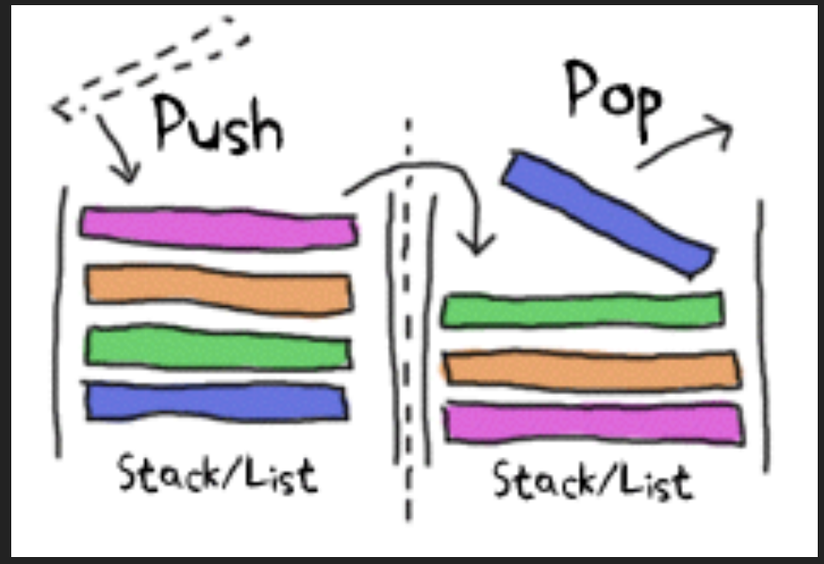
- implementation details?
  - gb_sets, ordsets and sofs all use the == operator to compare values
- sets (the module) uses the =:= operator, which means you can't necessarily switch over every implementation as you wish

DIRECTED GRAPHS

- mathematical theory that goes with it
  - digraph
    - manipulating edges and vertices, finding paths and cycles, etc
- digraph_utils
  - navigate a graph (postorder, preorder), testing for cycles, arborescences or trees, finding neighbors, and so on.
- sofs contatins converting functions

QUEUES

- double-ended FIFO
- mental separation to 3 interface (varying complexity)
  - Original API
  - Extended API
  - Okasaki API



QUEUES

- Original Api
  - at the base of the queue concept

```
new/0, in/2,  out/1
```

- Extended Api
  - introspection power and flexibility

```
get/1, peek/1, drop/1
```

QUEUES

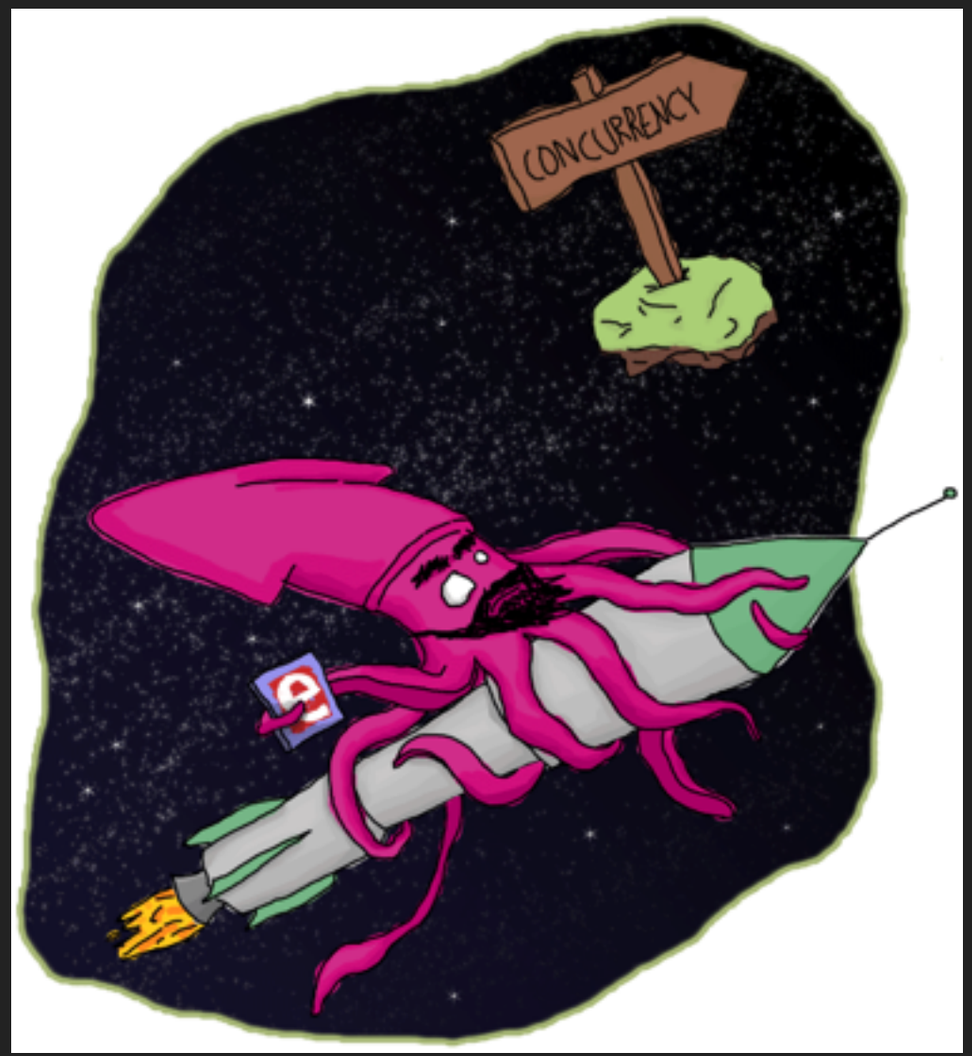
- okasaki API
  - The API provides operations similar to what was available in the two previous APIs, but some of the function names are written backwards and the whole thing is relatively peculiar.
- when to use
  - In cases where you can't just do all the reversing at once and elements are frequently added, the queue module is what you want

END OF SHORT VISIT

- recommand explore
  - http://www.erlang.org/doc/apps/stdlib/index.html
  - http://www.erlang.org/doc/applications.html

END OF SHORT VISIT

- concurrency and processes is where Erlang shines

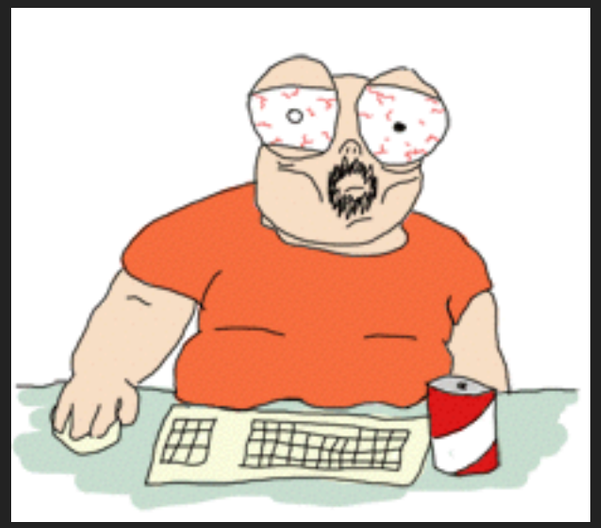


CHAPTER. 10

THE HITCHHIKER'S GUIDE TO CONCURRENCY

DON'T PANIC

- concurrency
  - many actors running independently, but not necessarily all at the same time
- parallelism
  - exactly same time
- doesn't seem to be any consensus on such definitions ..
  - don't be surprised !

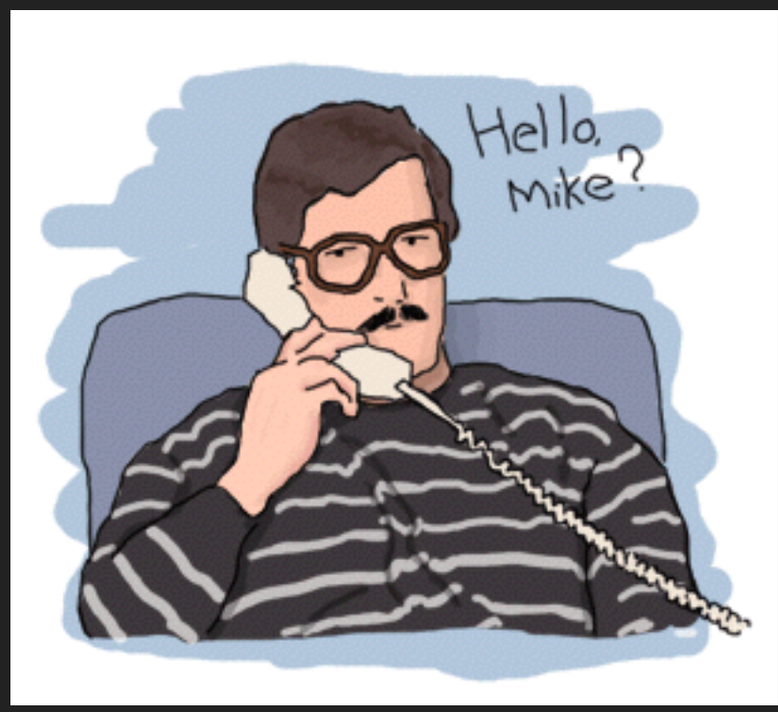


DON'T PANIC

- erlang's concurrency is already there when '80s (slice)
- nowadays, parallelism is on single computer thanks to industrial chips !
- don't drink kool aid
  - erlang symmetric multiprocessing is available since 2009

CONCEPT OF CONCURRENCY

- erlang's development boosted by telephone engineers feedbacks
  - processes-based concurrency / asynchronous message passing



CONCEPT OF CONCURRENCY

- scalability
  - processes which only reacted upon certain events
  - processes doing small computations, switching between them very quickly as events came through
  - very quickly started, very quickly destroyed
  - you didn't want to have things like process pools

CONCEPT OF CONCURRENCY

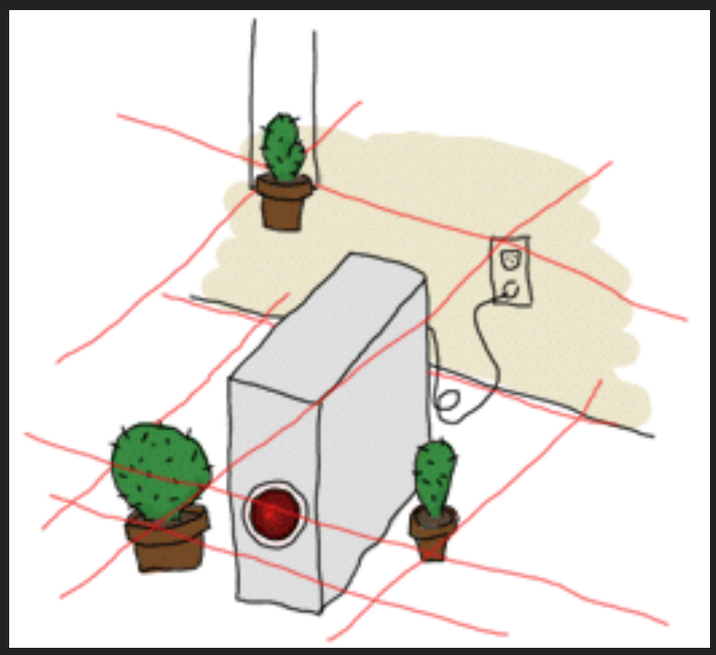
- scalability
  - be able to bypass your hardware's limitations
    - make batter or, add more H/W (cheaper) -> distribution can be useful as part of your language
- telephony applications needed a lot of reliability
  - forbid processes from sharing memory

CONCEPT OF CONCURRENCY

- fault-tolerance
  - erlang writer kept in mind that failure is common
  - so multiple processes with message passing was a good idea
    - intermittent or transient bugs
- A safe solution would be to make sure all crashes are the same as clean shutdowns
  - isolate process mem, avoiding lock

CONCEPT OF CONCURRENCY

- fault-tolerance
  - how about H/W fault?
  - someone's kicking? -> distribution



CONCEPT OF CONCURRENCY

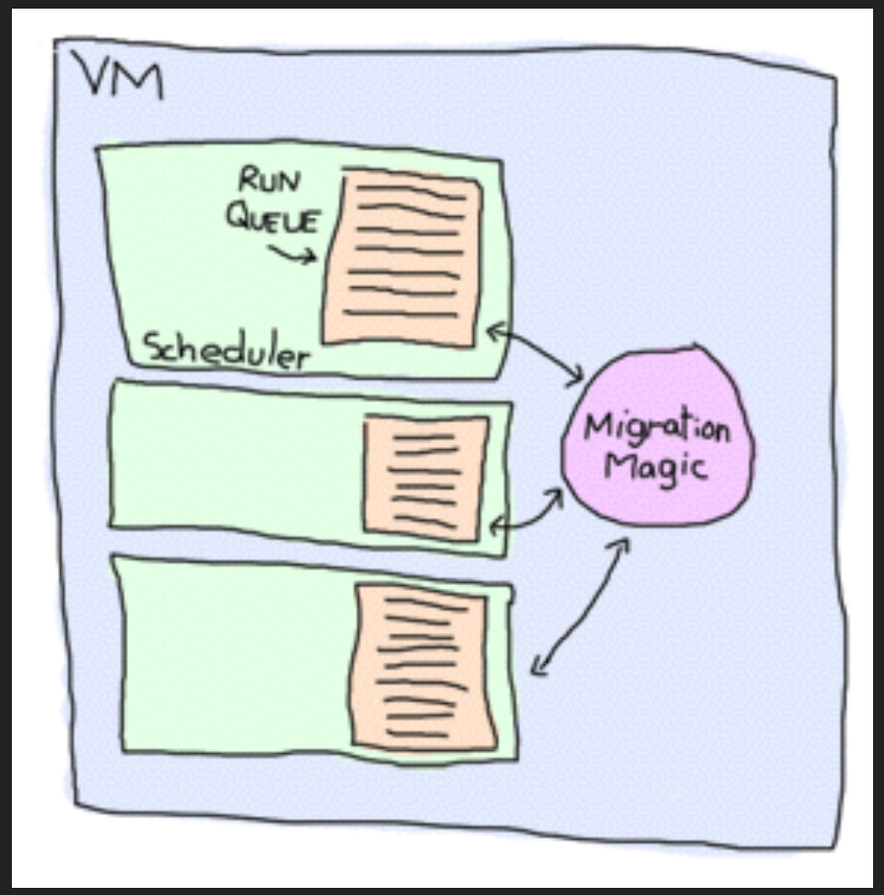
- fault-tolerance
  - distributions problem?
    - you can't assume that because a node (a remote computer) was there
- asynchornose message passing!
  - stored in a mailbox
  - you can use ack

CONCEPT OF CONCURRENCY

- implementation
  - OS cannot be trusted -> VM
  - Erlang's processes take about 300 words of memory each and can be created in a matter of microseconds

CONCEPT OF CONCURRENCY

- implementation
  - to handle potential processes your program could create, VM start one thread per core as scheduler
  - run queue
  - or list of erlang process

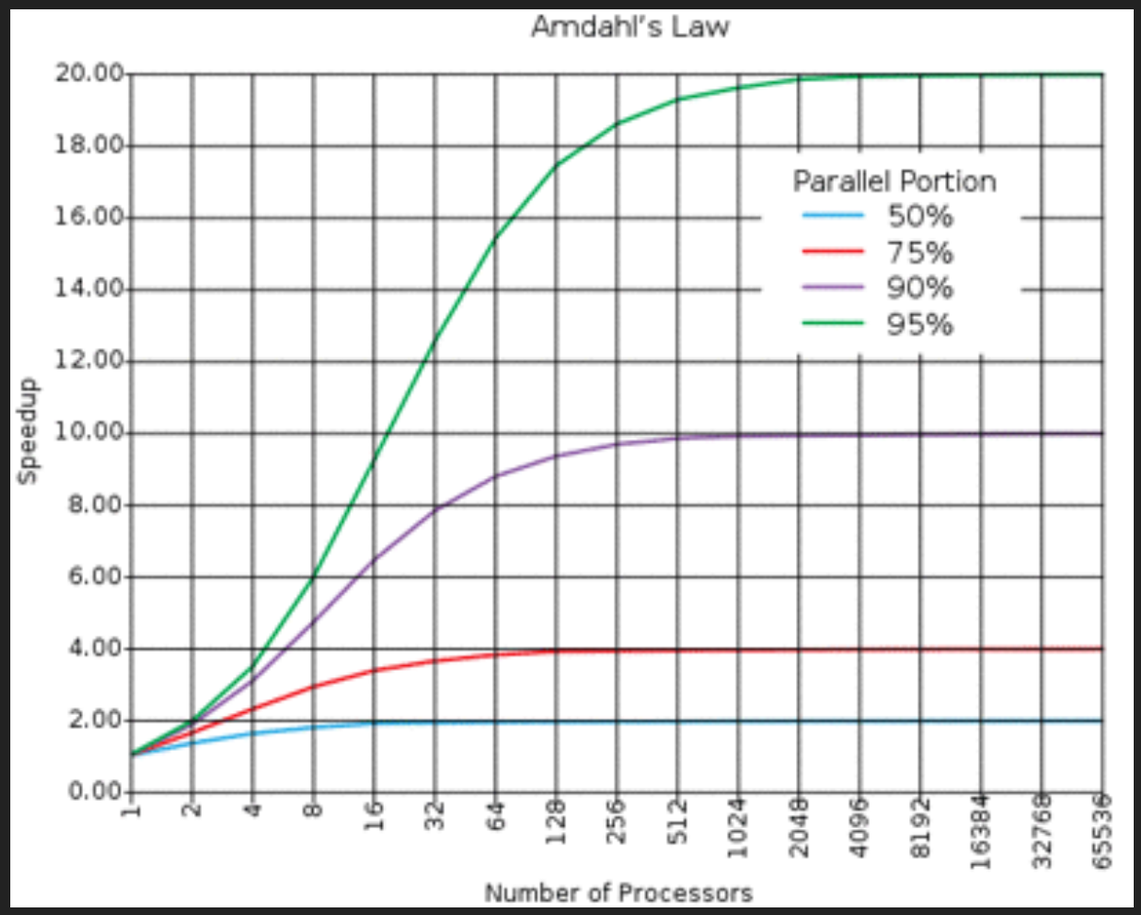


NOT ENTIERLY UNLIKE LINEAR SCALING

- difficulty of linear scaling is come from the nature (not from the erlang)
- Problems that scale very well are often said to be embarrassingly parallel
- IRC, ask... all these problems are usually about numerical algorithms with lots of data crunching
  - Erlang's embarrassingly parallel problems are present at a higher level

NOT ENTIERLY UNLIKE LINEAR SCALING

- Your parallel program only goes as fast as its slowest sequential part



NOT ENTIERLY UNLIKE LINEAR SCALING

- In some cases, going parallel will even slow down your application
  - This can happen whenever your program is 100% sequential, but still uses multiple processes.

SO LONG AND THANKS FOR ALL THE FISH!

- process is - just function (enough for now)

```
1> F = fun() -> 2 + 2 end.
   #Fun<erl_eval.20.67289768>
2> spawn(F).
   <0.44.0>
```

SO LONG AND THANKS FOR ALL THE FISH!

- process is - just function (enough for now)

```
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
  #Fun<erl_eval.6.13229925>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
  [<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,
  <0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]
2
1
4
3
5
..
```

SO LONG AND THANKS FOR ALL THE FISH!

- bang (!)

```
9> self() ! hello.
hello
```

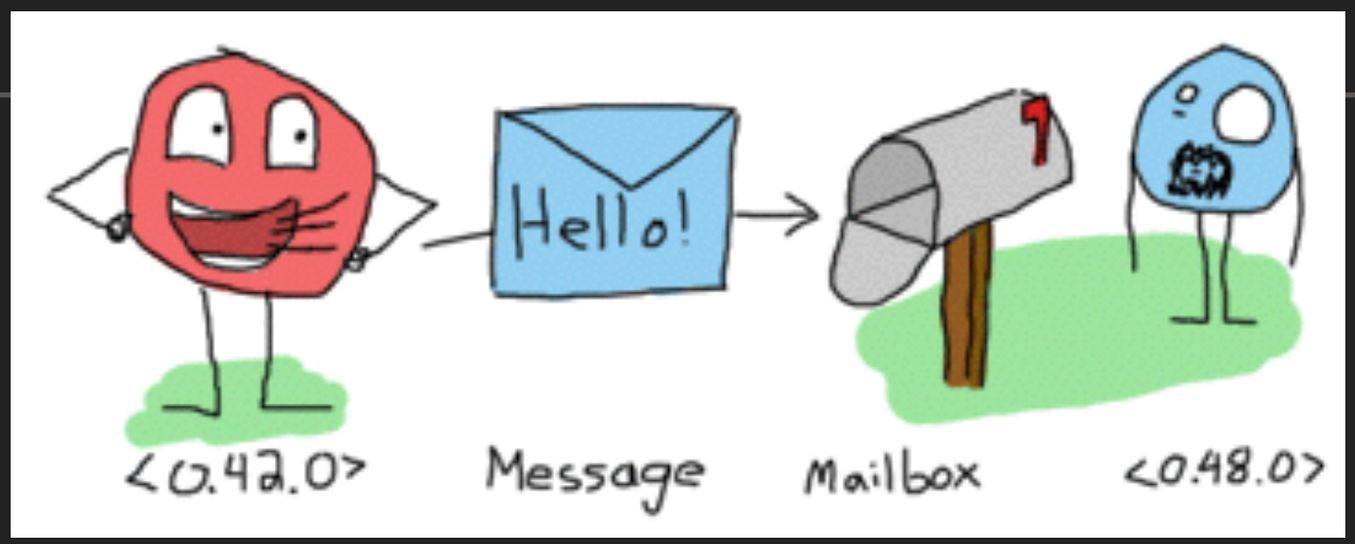
- second hello is return value of send operation

```
10> self() ! self() ! double.
double
```

SO LONG AND THANKS FOR ALL THE FISH!

- open the mailbox (just a shortcut that outputs received messages)

```
11> flush().
Shell got hello
Shell got double
Shell got double
ok
```



SO LONG AND THANKS FOR ALL THE FISH!

- receive

```
receive
  Pattern1 when Guard1 -> Expr1;
  Pattern2 when Guard2 -> Expr2;
  Pattern3 -> Expr3
end
```

SO LONG AND THANKS FOR ALL THE FISH!

- receive

```
-module(dolphins).
-compile(export_all).

dolphin1() ->
  receive
    do_a_flip ->
      io:format("How about no?~n");
    fish ->
      io:format("So long and thanks for all the fish!~n");
    _ ->
      io:format("Heh, we're smarter than you humans.~n")
  end.
```

SO LONG AND THANKS FOR ALL THE FISH!

- using it

```
11> c(dolphins).
  {ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
  <0.40.0>
13> Dolphin ! "oh, hello dolphin!".
  Heh, we're smarter than you humans.
  "oh, hello dolphin!"
14> Dolphin ! fish.
  fish
15>
```

SO LONG AND THANKS FOR ALL THE FISH!

- spawn/3
  - 1. hit receive statement, mail box is empty, so wait.
  - 2. pattern match
  - 3. run

SO LONG AND THANKS FOR ALL THE FISH!

- using it

```
8> f(Dolphin).
  ok
9> Dolphin = spawn(dolphins, dolphin1, []).
  <0.53.0>
10> Dolphin ! fish.
  So long and thanks for all the fish!
  fish
```

SO LONG AND THANKS FOR ALL THE FISH!

- send reply

```
dolphin2() ->
  receive
    {From, do_a_flip} ->
      From ! "How about no?";
    {From, fish} ->
      From ! "So long and thanks for all the fish!";
    _ ->
      io:format("Heh, we're smarter than you humans.~n")
end.
```

SO LONG AND THANKS FOR ALL THE FISH!

- send

```
11> c(dolphins).
  {ok,dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
  <0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
  {<0.32.0>,do_a_flip}
14> flush().
  Shell got "How about no?"
  ok
```

SO LONG AND THANKS FOR ALL THE FISH!

- don't be shut it down !

```
dolphin3() ->
  receive
    {From, do_a_flip} ->
      From ! "How about no?",
      dolphin3();
    {From, fish} ->
      From ! "So long and thanks for all the fish!";
    _ ->
      io:format("Heh, we're smarter than you humans.~n"),
      dolphin3()
  end.
```

SO LONG AND THANKS FOR ALL THE FISH!

```
15> Dolphin3 = spawn(dolphins, dolphin3, []).
  <0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
  {<0.32.0>,do_a_flip}
17> flush().
  Shell got "How about no?"
  Shell got "How about no?"
  ok
18> Dolphin3 ! {self(), unknown_message}.
  Heh, we're smarter than you humans.
  {<0.32.0>,unknown_message}
19> Dolphin3 ! Dolphin3 ! {self(), fish}.
  {<0.32.0>,fish}
20> flush().
  Shell got "So long and thanks for all the fish!"
  ok
```

SO LONG AND THANKS FOR ALL THE FISH!

- There you have it!