

Erlang

Fault Tolerant

The right concurrency model.
Error & exception handling done right
Good libraries for the hard stuff

Erlang

Maintainable

Dynamically typed.
Symbolic and transparent data structures
An interactive shell

Erlang

Scalable

The right concurrency model.
Good libraries for the hard stuff
Weird but efficient strings for I/O



ERTS & Performance

- Code size/size
- Scheduling
- Memory management
- GC

A Tuning Strategy

Work per unit of time
Memory per unit of time
Development per unit of time
Performance per unit of time
Power per unit of time



What is ERTS?

ERTS is the Erlang Runtime System.

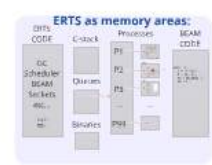
ERTS as source code:

See: [OTP]/erts/
emulator/
beam/
hipe/
etc/

ERTS as components:

The BEAM interpreter
The Scheduler
The Garbage Collector

Processes
HIPE
I/O



Sharing

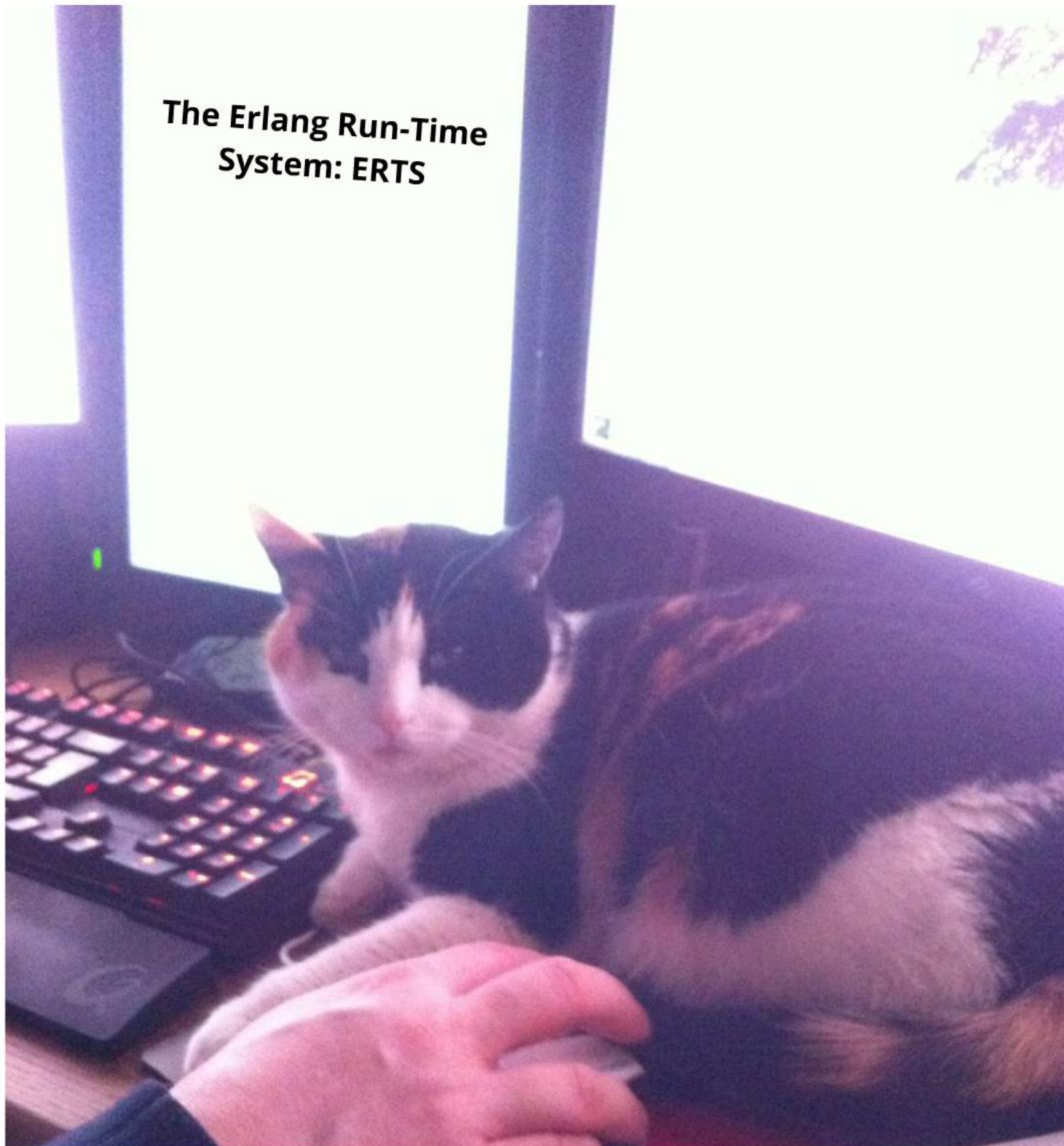


Lessons learned:

- Work per unit of time
- Memory per unit of time
- Development per unit of time
- Performance per unit of time
- Power per unit of time

Identification
Do a GC in a temp area
Check size
Allocate a normal heap area
reuse free slots

**The Erlang Run-Time
System: ERTS**



A scenic photograph of a sunset over a forest. The sun is low on the horizon, casting a warm, golden glow across the sky and reflecting on a body of water in the foreground. The trees are silhouetted against the bright light, and a lens flare is visible on the right side of the image.

Erlang

Fault Tolerant

The right concurrency model.
Error & exception handling done right
Good libraries for the hard stuff

A scenic background image showing a sunset over a forest. The sun is low on the horizon, casting a warm, golden glow through the trees. The sky is a mix of orange, yellow, and blue. The foreground is dark, with the silhouettes of trees and a body of water reflecting the light.

Erlang

Maintainable

Dynamically typed.

Symbolic and transparent data structures

An interactive shell

A scenic background image showing a sunset over a body of water, with a dense forest of tall trees in the foreground and middle ground. The sun is low on the horizon, creating a bright orange and yellow glow that reflects on the water and illuminates the sky. The trees are silhouetted against the bright light.

Erlang

Scalable

The right concurrency model.
Good libraries for the hard stuff
Weird but efficient strings for I/O

The right concurrency model

Lightweight processes
Message passing
Share nothing semantics
Don't stop the world GC
Monitors & Signals

A process is just memory



Error & exception handling done right



Dynamically & Strongly Typed
Symbolic and transparent data structures

Enables:
Hot code loading
Mutable heaps & stacks
Transparency of data
Garbage Collection



Remember
I don't have
opinions
this is all
facts.

The right concurrency model

Lightweight processes
Message passing
Share nothing semantics
Don't stop the world GC
Monitors & Signals

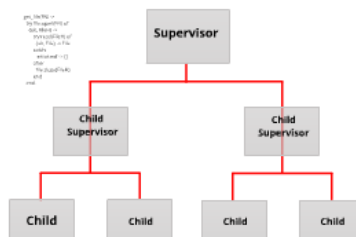
A Process is just memory



+ Preemptive multitasking implemented through reduction counting

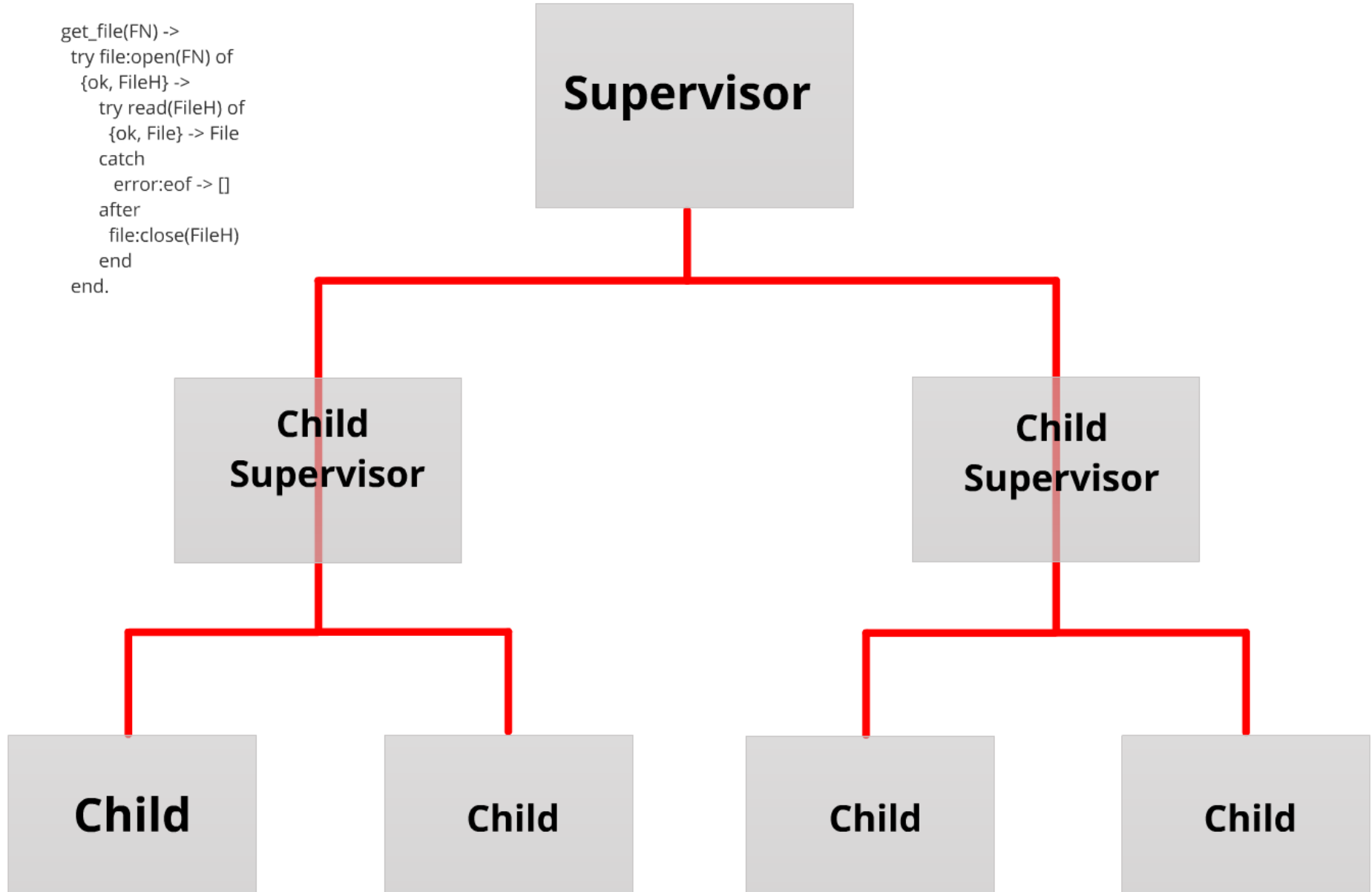


Error & exception handling done right



Error & exception handling done right

```
get_file(FN) ->  
try file:open(FN) of  
  {ok, FileH} ->  
    try read(FileH) of  
      {ok, File} -> File  
    catch  
      error:eof -> []  
    after  
      file:close(FileH)  
    end  
end.  
end.
```



Dynamically & Strongly Typed

Symbolic and transparent data structure

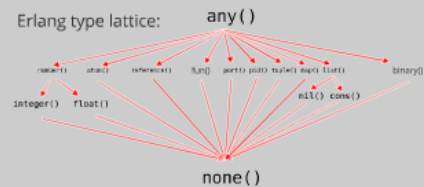
Enables:

Hot code loading

Movable heaps & stacks

Transparency of data

Garbage Collection



All values are tagged, some are boxed.



What is ERTS?

ERTS is the Erlang Runtime System.



SIMPLICITY

IT'S FOR SIMPLETONS

DIY.DESPAIR.COM

Processes

Conceptually: 4 memory areas and a pointer:

A Stack

A Heap

A Mailbox

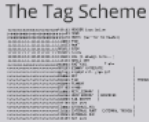
A Process Control Block

A PID

ERTS as memory areas:

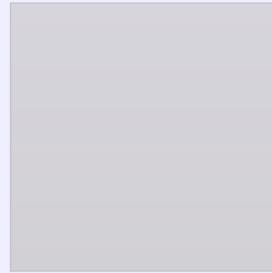
ERTS
CODE

GC
Scheduler
BEAM
Sockets
etc...



The Tag Scheme diagram shows a list of memory tags and their corresponding values, such as 0 for nil, 1 for true, and various pointers to memory blocks.

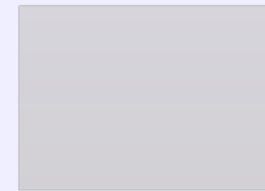
C-stack



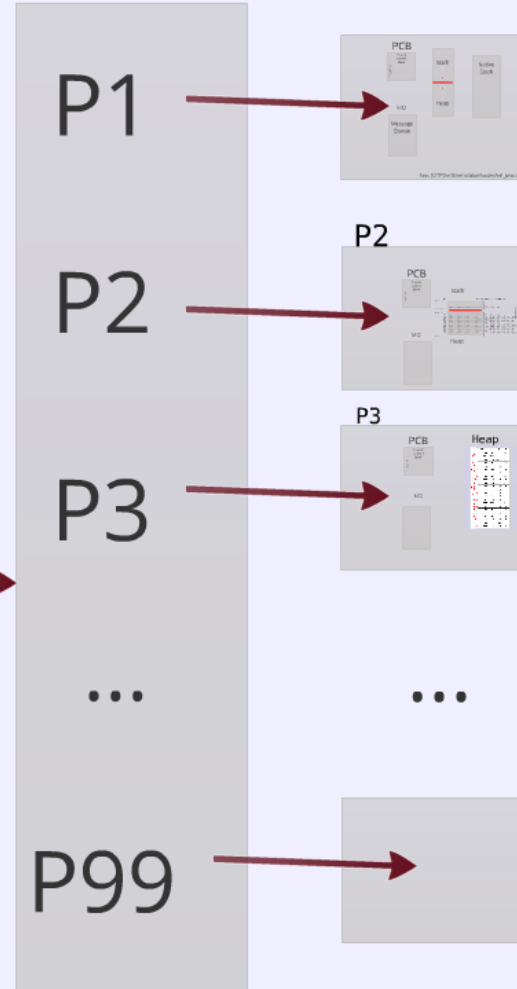
Queues



Binaries



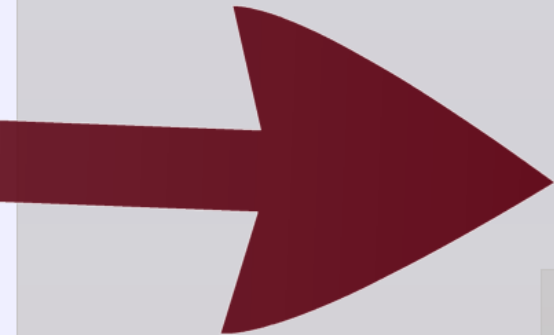
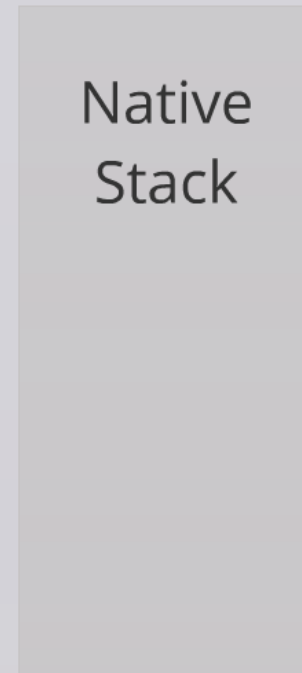
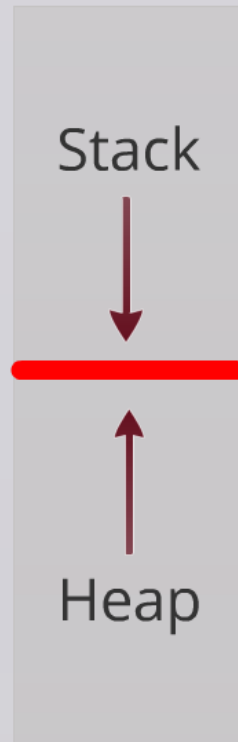
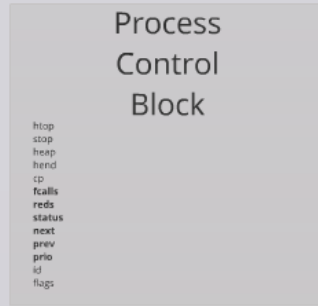
Processes



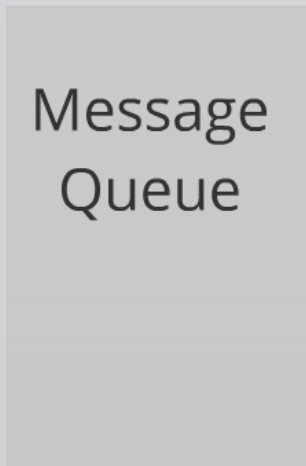
BEAM
CODE

```
p2() ->
  L = "Hello",
  T = {L, L},
  P3 = mk_proc(),
  P3 ! T.
```


PCB



MQ



Process Control Block

htop
stop
heap
hend
cp
fcalls
reds
status
next
prev
prio
id
flags

The Tag Scheme

aaaaaaaaaaaaaaaaaaaaaaaaaatttt00	HEADER (see below)			
ppppppppppppppppppppppppppppppppp01	CONS			
ppppppppppppppppppppppppppppppppp10	BOXED (pointer to header)			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011	PID			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111	PORT			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011	ATOM			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011	CATCH			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011	NIL (i always zero...)			
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111	SMALL_INT			
aaaaaaaaaaaaaaaaaaaaaaaaa000000	ARITYVAL	Tuple		
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv000100	BINARY_AGGREGATE		 	
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv001x00	BIGNUM with sign bit			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv010000	REF			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv010100	FUN			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv011000	FLONUM			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv011100	EXPOR			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv100000	REFC_BINARY	 		
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv100100	HEAP_BINARY			BINARIES
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv101000	SUB_BINARY			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv101100	Not used			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv110000	EXTERNAL_PID		 	
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv110100	EXTERNAL_PORT	EXTERNAL THINGS		
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv111000	EXTERNAL_REF			
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv111100	Not used			

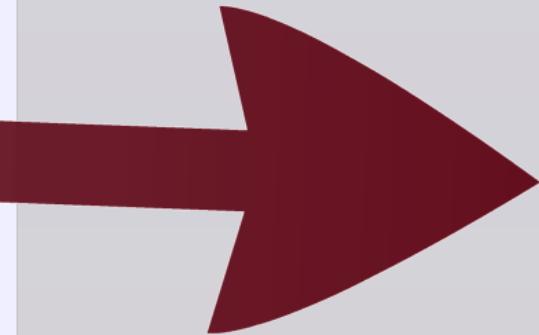
An example

The string "Hello", i.e. the list
[104, 101, 108, 108, 111]

PCB

Process
Control
Block

htop
stop
heap
hend
cp
fcalls
reds
id
flags
next
prev
...

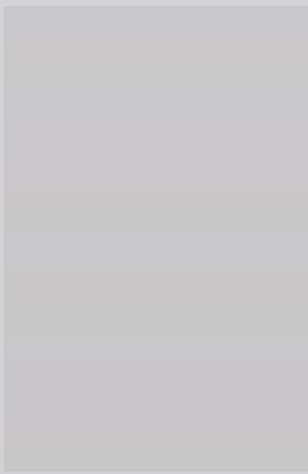


MQ

Stack

	ADR	BINARY	VALUE	DESCRIPTION
hend ->			
			
stop ->		00000000 00000000 00000000 10000001	128 + list tag	-----+
htop ->			
132		00000000 00000000 00000000 01111001	120 + list tag	-----+ -+
128		00000000 00000000 00000110 10001111	(H) 104 bsl 4 + small int tag	<+
124		00000000 00000000 00000000 011110001	112 + list tag	-----+ -+
120		00000000 00000000 00000110 01011111	(e) 101 bsl 4 + small int tag	<----+
116		00000000 00000000 00000000 01110001	112 + list tag	-----+ -+
112		00000000 00000000 00000110 11001111	(l) 108 bsl 4 + small int tag	<-----+
108		00000000 00000000 00000000 01110001	96 + list tag	-----+ -+
104		00000000 00000000 00000110 11001111	(l) 108 bsl 4 + small int tag	<-----+
100		11111111 11111111 11111111 11111011	NIL	
96		00000000 00000000 00000110 11111111	(o) 111 bsl 4 + small int tag	<-----+
heap ->			

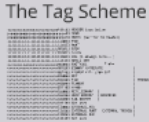
Heap



ERTS as memory areas:

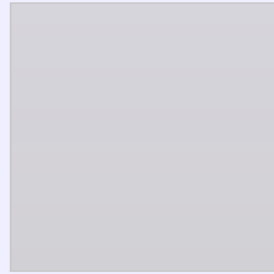
ERTS
CODE

GC
Scheduler
BEAM
Sockets
etc...



The Tag Scheme diagram shows a memory layout with various fields and pointers, including 'tag', 'value', and 'next' fields, illustrating the structure of memory tags in the BEAM system.

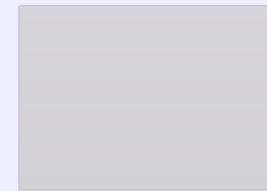
C-stack



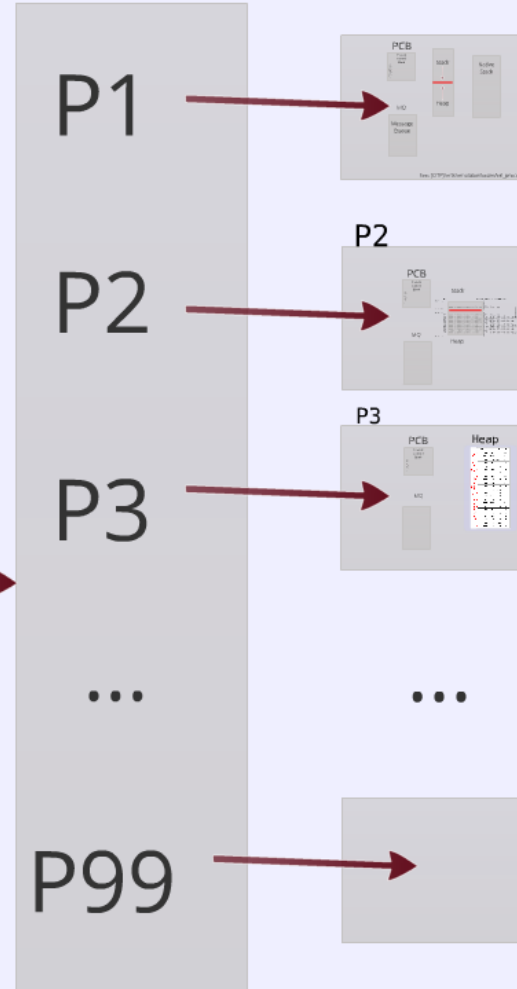
Queues



Binaries



Processes



BEAM
CODE

```
p2() ->  
L = "Hello",  
T = {L, L},  
P3 = mk_proc(),  
P3 ! T.
```


BEAM

- Garbage Collecting-
 - Reduction Counting-
 - Non-preemptive-
 - Directly Threaded-
 - Register-
 - Virtual-
- Machine



Memory Management:

Garbage Collection

- On the Beam level the code is responsible for:
 - checking for stack and heap overrun.
 - allocating enough space
- "test_heap" will check that there is free heap space.
- If needed the instruction will call the GC.
- The GC might call lower levels of the memory subsystem to allocate or free memory as needed.

Scheduling:

Non-preemptive, Reduction counting

- Each function call is counted as a reduction
- Beam does a test at function entry: if (reds < 0) yield
- A reduction should be a small work item
- Loops are actually recursions, burning reductions

A process can also yield in a receive.

Dispatch: Directly Threaded Code

The dispatcher finds the next instruction to execute.

I: 0x1000

```
#define Arg(N) (Eterm *) I[(N)+1]
#define Goto(Rel) goto *((void *)Rel)
```

External beam format:

```
{move,{x,0},{x,1}}.
{move,{y,0},{x,0}}.
{move,{x,1},{y,0}}.
```

Loaded code*:

```
0x1000: 0x3000
0x1004: 0x0
0x1008: 0x1
0x100c: 0x3200
0x1010: 0x1
0x1014: 0x1
0x1018: 0x3100
0x101c: 0x1
0x1020: 0x1
```

beam_emu.c **:

```
OpCase(move_xx): {
0x3000: x(Arg(1)) = x(Arg(0));
      | += 3;
      Goto(*I);
}
OpCase(move_yx): {
0x3200: x(Arg(1)) = y(Arg(0));
      | += 3;
      Goto(*I);
}
OpCase(move_xy): {
0x3100: y(Arg(1)) = x(Arg(0));
      | += 3;
      Goto(*I);
}
```

*This is a lie... beam actually rewrites the external format to different internal instructions....

** This is another lie... beam actually rewrites the external format to different internal instructions....

BEAM

- Garbage Collecting-
- Reduction Counting-
- Non-preemptive-
- Directly Threaded-
- Register-
- Virtual-

-Machine



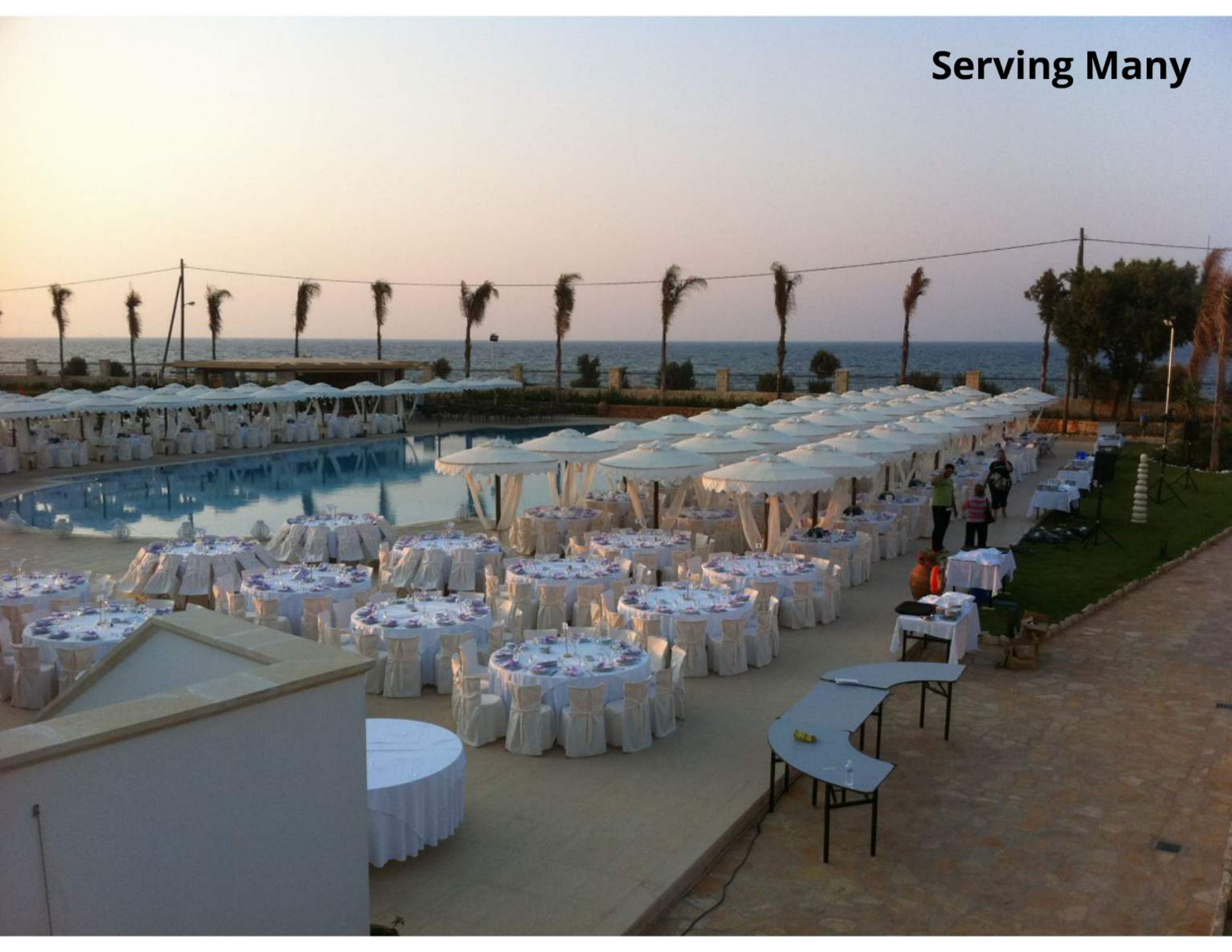
A War Story



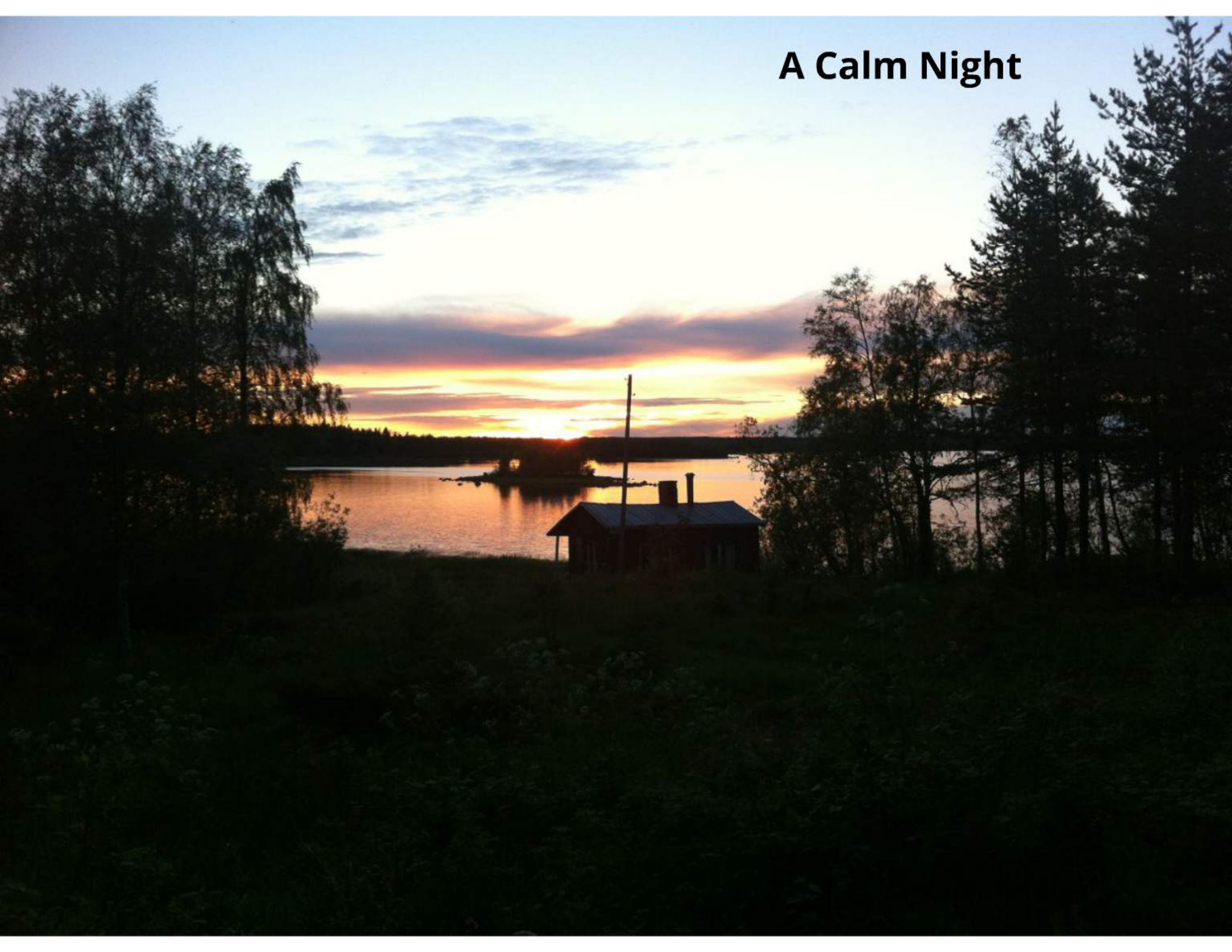
Serving a Few



Serving Many



A Calm Night





Where Did the Slave Node Go?



A scheduler went to sleep



Several Schedulers Went to Sleep

Last Live Scheduler
Tries term_to_binary



Heart Goes For the Kill



BEAM

- Garbage Collecting-
- Reduction Counting-
- Non-preemptive-
- Directly Threaded-
- Register-
- Virtual-

-Machine

```

1= compileFile(world, ['S', binary])

```

Beam Instructions

An Added Example

```

1= compileFile(world, ['S', binary])

```

BEAM is a register machine.
It has two sets of registers:
x and y

x registers are caller save
and arguments.
y registers are callee save
and actually the stack.

See: [DTP]erts/emulator/beam/beam_emul.c

You can look at beam code by giving
the 'S' flag to the compiler:

```

c(test, ['S']).

```

The Scheduler



Process State
Reductions
Que Handling
Timing wheels

Possible Problems

```

Should I be worried? No
Do I need to know about this? No
What should I do? I don't know what to do

```

erl_process.c schedule()

Lukas: "It is quite short and not hard to understand if you know C".

```

beam_emul.c
process_main()
execute process
if (wait)
  call schedule()

```

1. Update reduction counters
2. Check triggered timers
3. If check_balance_reductions > 4,000,000 check balance
4. Possibly migrate processes+ports
5. Execute scheduler work (load, free, trace, etc)
6. If function_calls > 4000 check IQ, update time
7. Execute 1 to N ports for 20000 rebs

Reduction count problems

BIFs uses an arbitrary amount of reductions.

A return does not use any reductions.

NIFs uses an arbitrary amount of reductions.

Load Balancing

Load balancing operations are calculated when a scheduler has done 4,000,000 reductions.

Processes will normally migrate towards lower schedulers if there is no overload.

If a scheduler is overloaded processes are evicted to other schedulers.

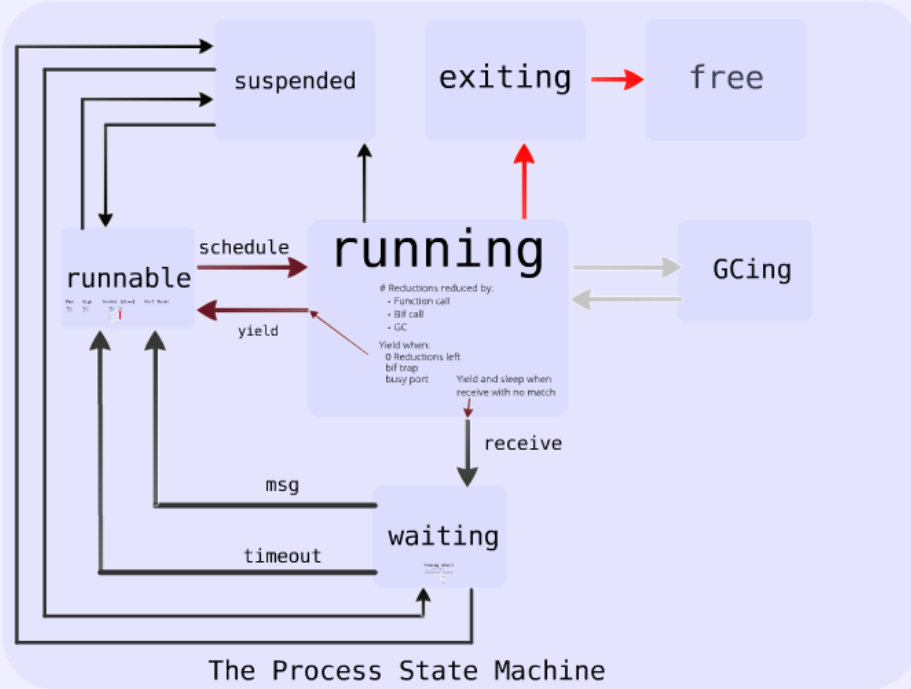
If reduction counting is messed up, starvation might occur.

Use: +swt to wake up sleeping schedulers.

The Garbage Co

One Scheduler Per Core

Cores	Schedulers	Running	Ready Q
a	1	1	2 3
b	2	4	5
c	3	6	
d	4	7	



Process State Reductions Que Handling Timing wheels

Possible Problems

Priority Inversion

Should I be worried?

No

Do I need to know about this?

No

What can I do?

Don't mess with priorities

running

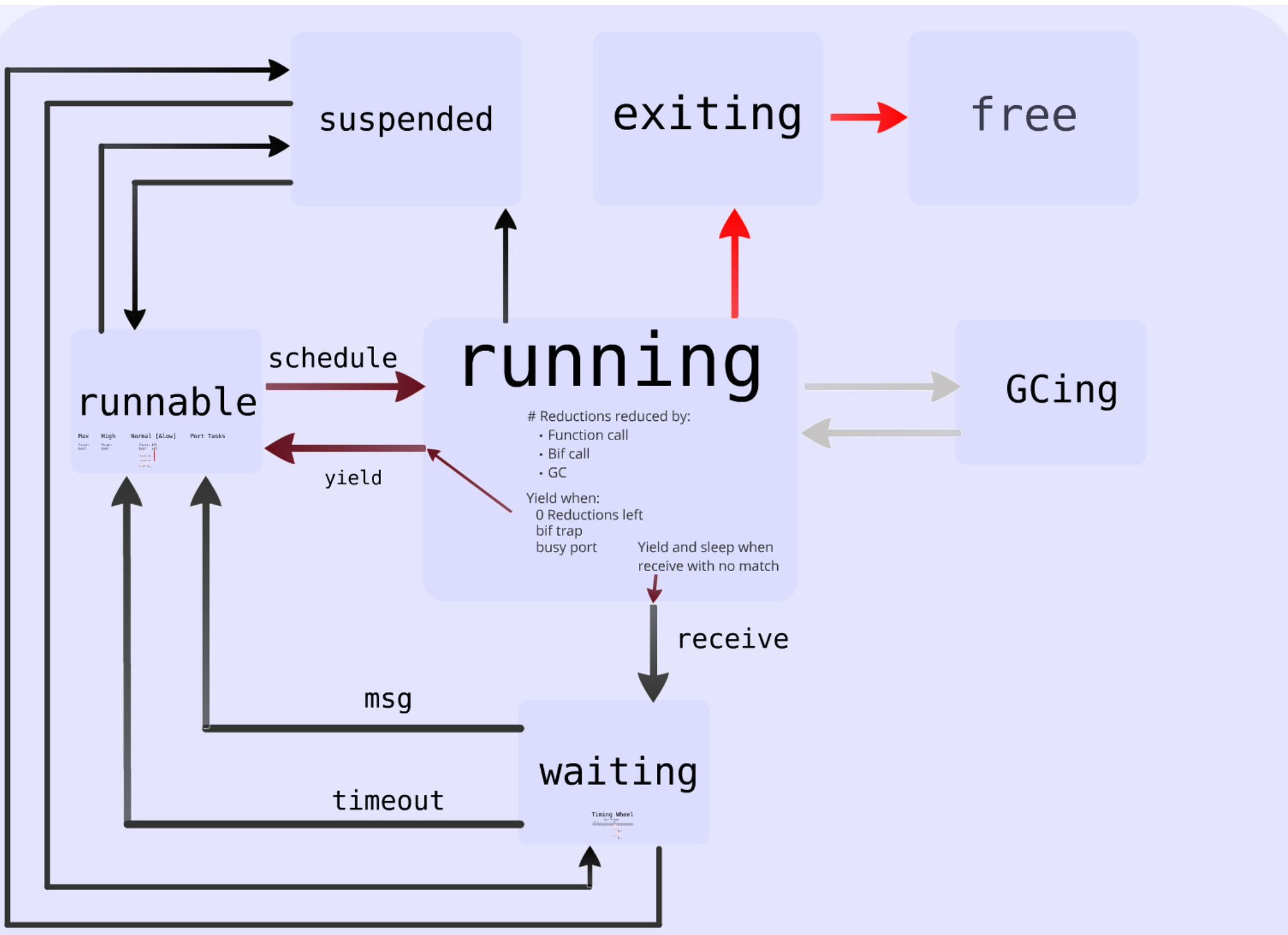
Reductions reduced by:

- Function call
- Bif call
- GC

Yield when:

0 Reductions left
bif trap
busy port

Yield and sleep when
receive with no match



The Process State Machine

runnable

Max

High

Normal [`&low`]

Port Tasks

First:
Last:

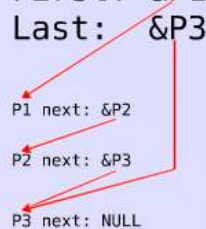
First:
Last:

First: `&P1`
Last: `&P3`

`P1 next: &P2`

`P2 next: &P3`

`P3 next: NULL`



Normal [&low]

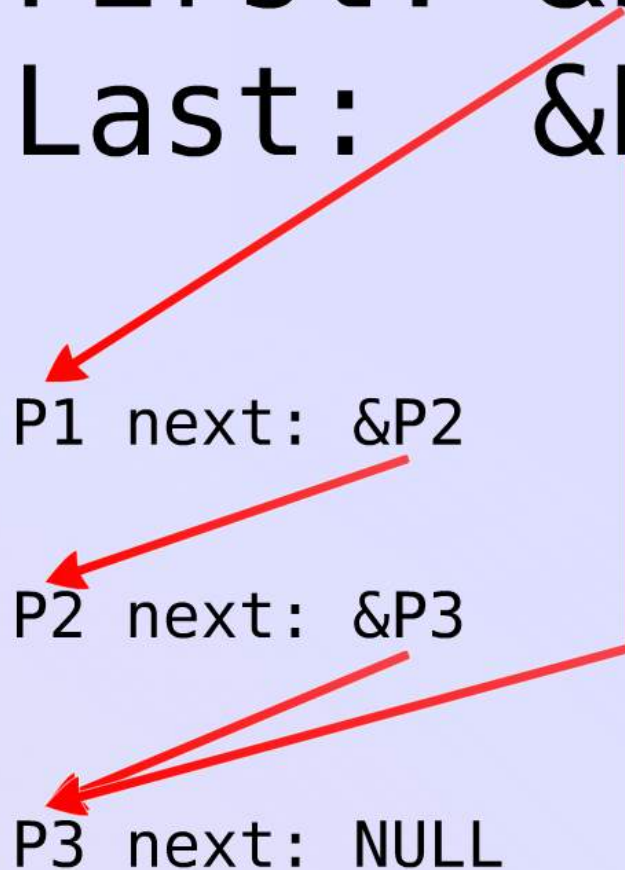
First: &P1

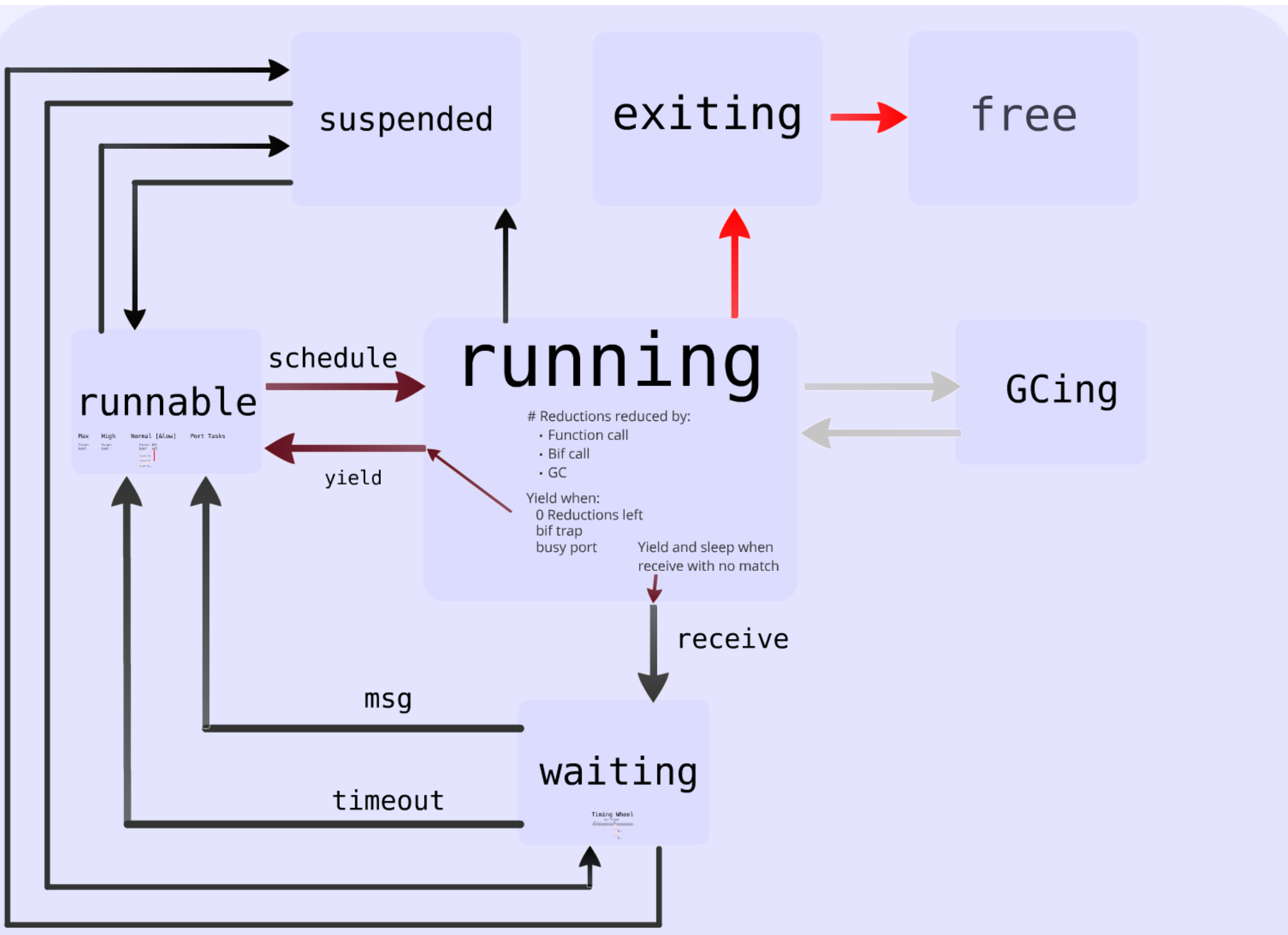
Last: &P3

P1 next: &P2

P2 next: &P3

P3 next: NULL

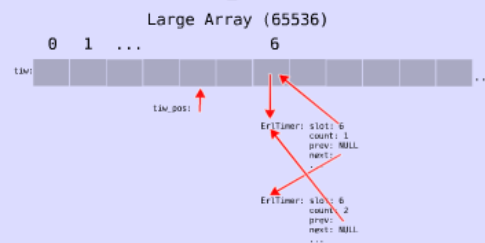




The Process State Machine

waiting

Timing Wheel



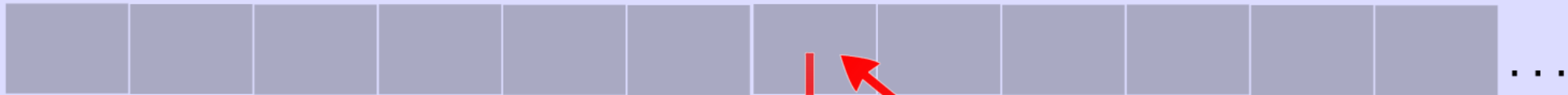
Timing Wheel

Large Array (65536)

0 1 ...

6

tiw:

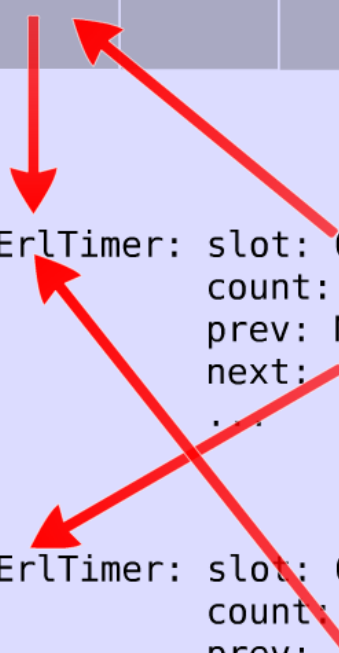


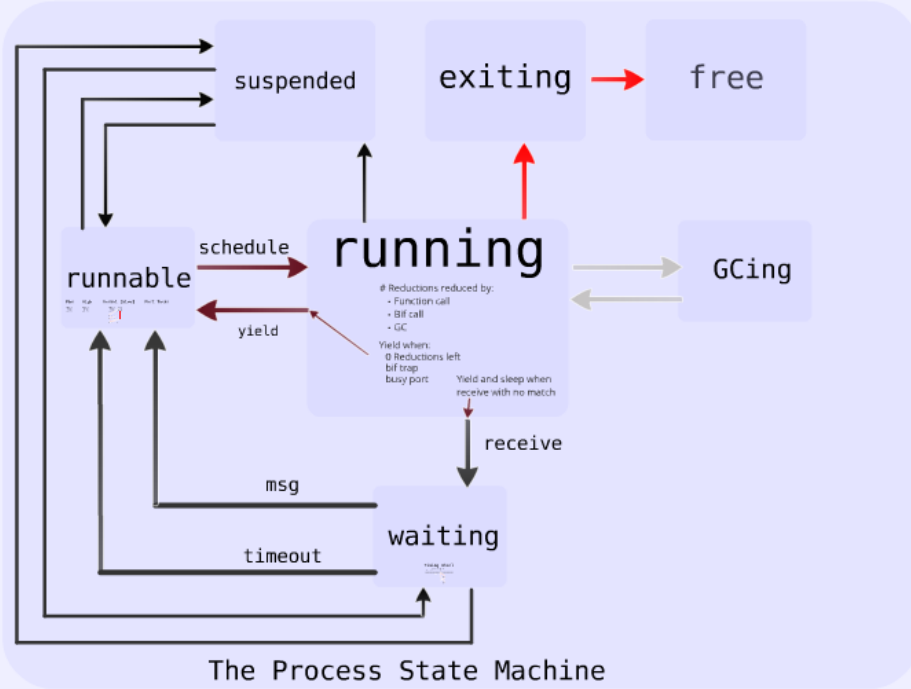
tiw_pos:



ErlTimer: slot: 6
count: 1
prev: NULL
next:
...

ErlTimer: slot: 6
count: 2
prev:
next: NULL
...





Process State Reductions Que Handling Timing wheels

Possible Problems

Priority Inversion

Should I be worried?

No

Do I need to know about this?

No

What can I do?

Don't mess with priorities

erl_process.c schedule()

Lukas: "It is quite short and not hard to understand if you know C".

560 lines

beam_emu.c
process_main()

Execute process
till yield.
call schedule()

schedule()

1. Update reduction counters
2. Check triggered timers
3. If `check_balance_reds > 4,000,000` check balance
4. Possibly migrate processes+ports
5. Execute scheduler work (load, free, trace, etc)
6. If `function_calls > 4000` check IO, update time
7. Execute 1 to N ports for 2000 redds

(More or less stolen from Lukas presentation)

Load Balancing

Load balancing operations are calculated when a scheduler has done 4,000,000 reductions.

Processes will normally migrate towards lower schedulers if there is no overload.

If a scheduler is overloaded processes are evicted to other schedulers.

If reduction counting is messed up, starvation might occur.

Use: +sfwi to wake up sleeping schedulers.

Reduction count problems

BIFs uses an arbitrary amount of reductions.

Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do? Fix the BIF ;) Use small data sets, add a call to erlang:bump_reductions()

A return does not use any reductions.

Should I be worried? No
Do I need to know about this? Probably not
What can I do? Use tail recursion, don't have insanely long callchains.

NIFs uses an arbitrary amount of reductions.

Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do? Don't use NIFs ;) Make sure your NIFs are yielding and using reductions. Wait for "dirty schedulers".

in a counter-pro

n arbitrary amo

Should I be worried?

Yes.

Do I need to know about this?

Yes.

What can I do?

Fix the BIF ;)

Use small data sets, add a call to `erlang:bump_reductions()`

es not use any

Reduction count problems

BIFs uses an arbitrary amount of reductions.

Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do? Fix the BIF ;)
Use small data sets, add a call to erlang.bump_reducers()

A return does not use any reductions.

Should I be worried? No
Do I need to know about this? Probably not
What can I do? Use tail recursion,
don't have insanely long callchains.

NIFs uses an arbitrary amount of reductions.

Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do? Don't use NIFs ;)
Make sure your NIFs are yielding and using reductions.
Wait for "dirty schedulers".

es not use a

Should I be worried?

No

Do I need to know about this?

Probably not

What can I do?

Use tail recursion,
don't have insanely long callchains.

REDUCTIONS!

Should I be worried? Yes.
Do I need to know about this?
What can I do? Fix the BIF ;)
Use small data sets, add a call to erlang:bump_reductions()

A return does not use any reductions

Should I be worried? No
Do I need to know about this? Probably not
What can I do? Use tail recursion,
don't have insanely long callchains.

NIFs uses an arbitrary amount of reductions.

Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do? Don't use NIFs ;)
Make sure your NIFs are yielding and using reductions.
Wait for "dirty schedulers".

ERTS as components:

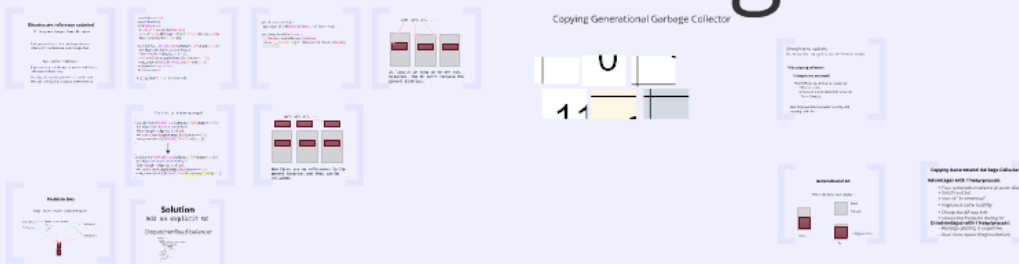
The BEAM interpreter



The Scheduler



The Garbage Collector



Processes

Conceptually: 4 memory areas and a pointer:

- A Stack
- A Heap
- A Mailbox
- A Process Control Block
- A PID

HiPE

I/O

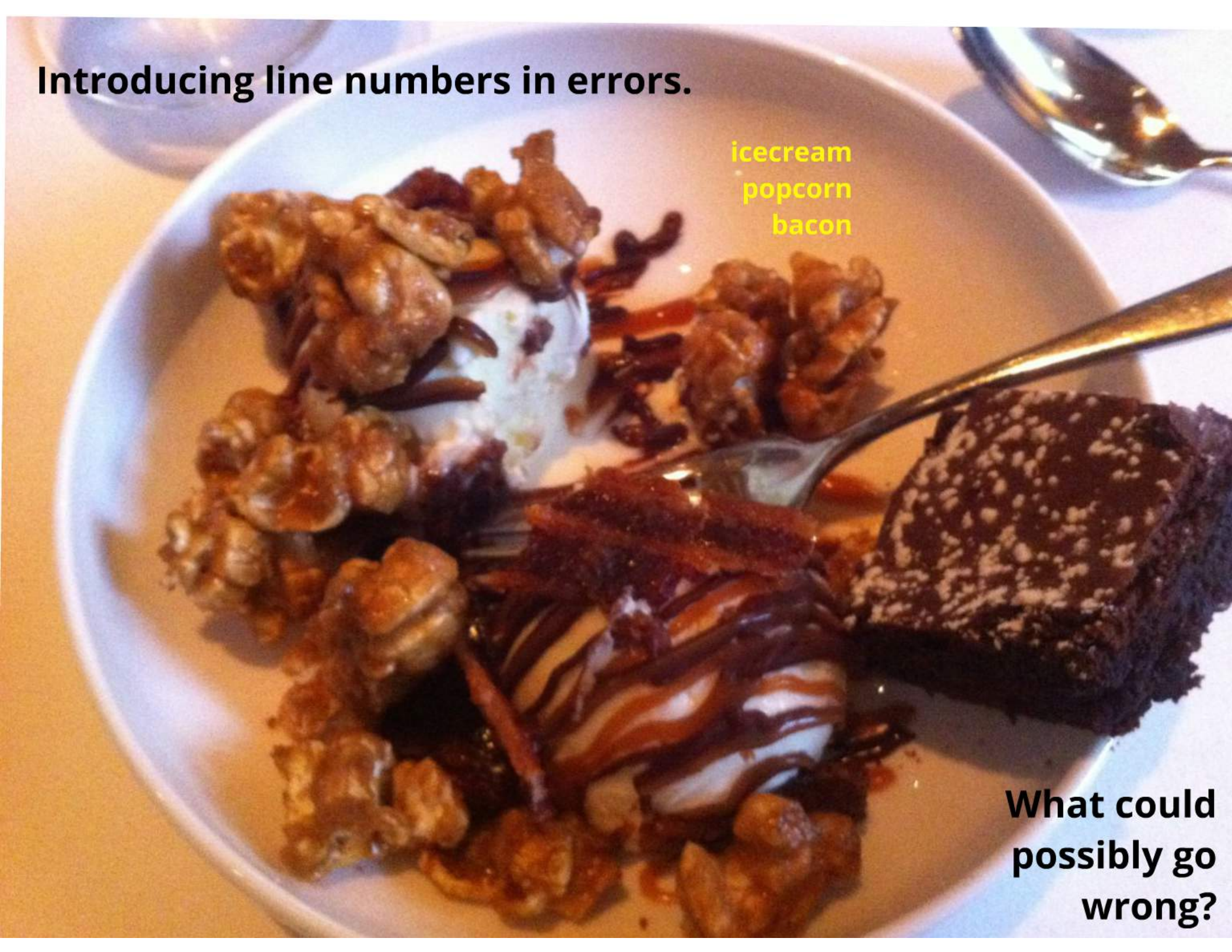


Another War Story

Introducing line numbers in errors.

icecream
popcorn
bacon

**What could
possibly go
wrong?**



Copying Generational Garbage Collector

Advantages with 1 heap/process:

- + Free reclamation when a process dies
- + Small root set
- + Sort of "Incremental"
- + Improved cache locality
- + Cheap stack/heap test
- + Small extra footprint during GC

Disadvantages with 1 heap/process:

- Message passing is expensive
- Uses more space (fragmentation)

Lessons learned:

- ERTS - the Erlang RunTime System is the defacto standard implementation of Erlang
- Each process has its own stack and heap
- The Erlang VM, BEAM, executes the Erlang code
- Process scheduling is controled by reduction count
- GC is local to a process
- GC is generational and copying





QUESTIONS?

The right people
Bright
Passionate
Get things
done



Erlang the language

```
% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional
Single-assignment
Dynamically typed

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

Concurrent
Distributed
Message passing

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

No sharing

Automatic Memory Management (GC)

Soft real-time

Fault tolerant

Open Source



```
%% File: hello.erl
```

```
-module(hello).
```

```
-export([run/0]).
```

```
run() -> io:format("Hello, World!\n").
```


Erlang the language

```
% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional
Single-assignment
Dynamically typed

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Concurrent
Distributed
Message passing

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

No sharing

Automatic Memory Management (GC)

Soft real-time

Fault tolerant

Open Source

```
F1 = fun () -> 42 end.
```

```
42 = F1().
```

```
F2 = fun (X) -> X + 1 end.
```

```
11 = F2(10).
```

```
F3 = fun (X, Y) ->
```

```
    {X, Y, Z}
```

```
end.
```

```
F4 = fun ({foo, X}, A) ->
```

```
    A + X*Y;
```

```
    ({bar, X}, A) ->
```

```
        A - X*Y;
```

```
    (_, A) ->
```

```
        A
```

```
end.
```

```
F5 = fun f/3
```

```
F6 = fun mod:f/3
```

Erlang the language

```
% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional
Single-assignment
Dynamically typed

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

Concurrent
Distributed
Message passing

```
1> erl -run hello run.
Hello, World!
```

```
1> erl -run hello run.
Hello, World!
```

No sharing

Automatic Memory Management (GC)

Soft real-time

Fault tolerant

Open Source


```
(foo@frodo)1> X=42.
```

```
42
```

```
(foo@frodo)2> X.
```

```
42
```

```
(foo@frodo)3> X=43.
```

```
** exception error: no match of right hand  
side value 43
```

```
(foo@frodo)4>
```

Erlang the language

```
%! File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional
Single-assignment
Dynamically typed

```
%% Example 1: Simple function
f(1) -> 1.
f(2) -> 2.
f(3) -> 3.
f(4) -> 4.
f(5) -> 5.
```

```
%% Example 2: Pattern matching
f(1) -> 1.
f(2) -> 2.
f(3) -> 3.
f(4) -> 4.
f(5) -> 5.
```

```
%% Example 3: List processing
f([]) -> [].
f([_ | _]) -> f([_]).
```

Concurrent
Distributed
Message passing

```
%% Example 4: Concurrent execution
f(1) -> 1.
f(2) -> 2.
f(3) -> 3.
f(4) -> 4.
f(5) -> 5.
```

```
%% Example 5: Distributed execution
f(1) -> 1.
f(2) -> 2.
f(3) -> 3.
f(4) -> 4.
f(5) -> 5.
```

No sharing

Automatic Memory Management (GC)

Soft real-time

Fault tolerant

Open Source

```
(foo@frodo)4> F= fun(X,Y) -> X + Y end.
```

```
#Fun<erl_eval.12.113037538>
```

```
(foo@frodo)5> F(1,2).
```

```
3
```

```
(foo@frodo)6> F(1.0, 2).
```

```
3.0
```

```
(foo@frodo)7> F("1", 2).
```

```
** exception error: bad argument in an arithmetic expression  
   in operator +/2  
   called as "1" + 2
```


Erlang the language

```
% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional
Single-assignment
Dynamically typed

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Concurrent
Distributed
Message passing

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

No sharing

Automatic Memory Management (GC)

Soft real-time

Fault tolerant

Open Source

Demo 1

```
%%% DEMO 1
%%% Show code
cd("C:/Users/happi").
S=demo:start_server().
W = demo:start_worker(S).
W2 = demo:start_worker(S).
S ! {broadcast, hi}.
W ! terminate.
exit(W2, kill).
S ! {broadcast, hi}.
exit(S, kill).
S ! {broadcast, hi}.
```


Easy to make fault-tolerant systems.

- Erlang was designed from the ground up with the purpose of making it easy to develop fault-tolerant systems.
- Erlang was developed by Ericsson with the telecom market in mind.
- Erlang supports processes, distributed systems, advanced exception handling, and signals.
- Erlang comes with OTP-libraries (Open Telecom Platform), e.g. supervisors and generic servers.

Low maintenace easy upgrade

Hot code loading.

Distribution.

Interactive shell.

Simple module system.

No shared state.

Virtual machine.

Network programming is easy

Distributed Erlang solves many network programming needs.

Setting up a simple socket protocol is a breeze.

The binary- (and now bit-) syntax makes parsing binary protocols easy.

There are simple but powerful libraries for HTTP, XML, XML-RPC and SOAP.

Ability to leverage multi-core

The concept of processes is an integral part of Erlang.

No shared memory -- easier to program.

The Erlang Virtual machine (BEAM) has support for symmetric multiprocessing.

“Each year your
sequential programs will
go slower.
Each year your
concurrent programs will
go faster.” -- Joe Armstrong

Rapid development

- Automatic memory management.
- Symbolic constants (atoms).
- An interactive shell.
- Dynamic typing.
- Simple but powerful data types.
- Higher order functions and list comprehensions.
- Built in (distributed) database.

God way to get great programmers.



Phil Lennart SPJ John

Nice paradox:

The lack of Erlang programmers makes it easier for us to find great programmers.

There are many great C and Java programmers, I'm sure, but they are hidden by hordes of mediocre programmers.

Programmers who know a functional programming language are often passionate about programming.

™

Passionate programmers makes **Great Programmers**