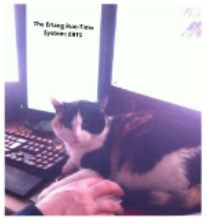


What is ERTS?

ERTS is the Erlang Runtime System.

ERTS as source code:

See: [OTP]/erts/

emulator/
beam/
hipe/
etc/

ERTS as components:

The BEAM interpreter

Processes

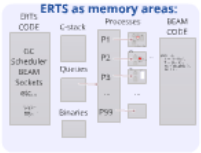
The Scheduler

HiPE

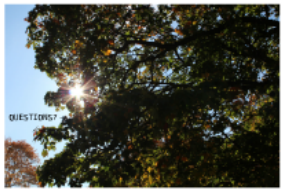
The Garbage Collector

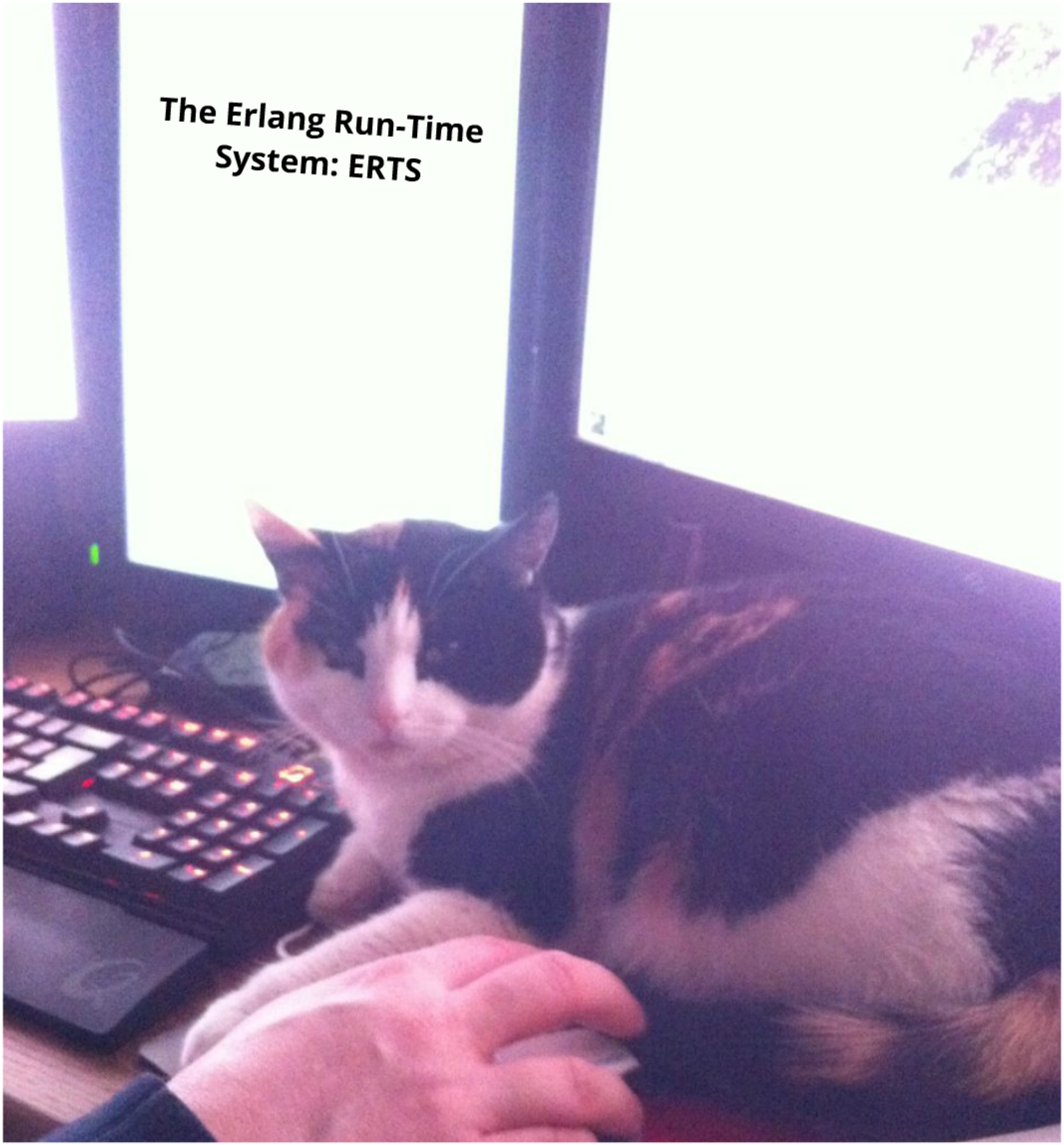
I/O

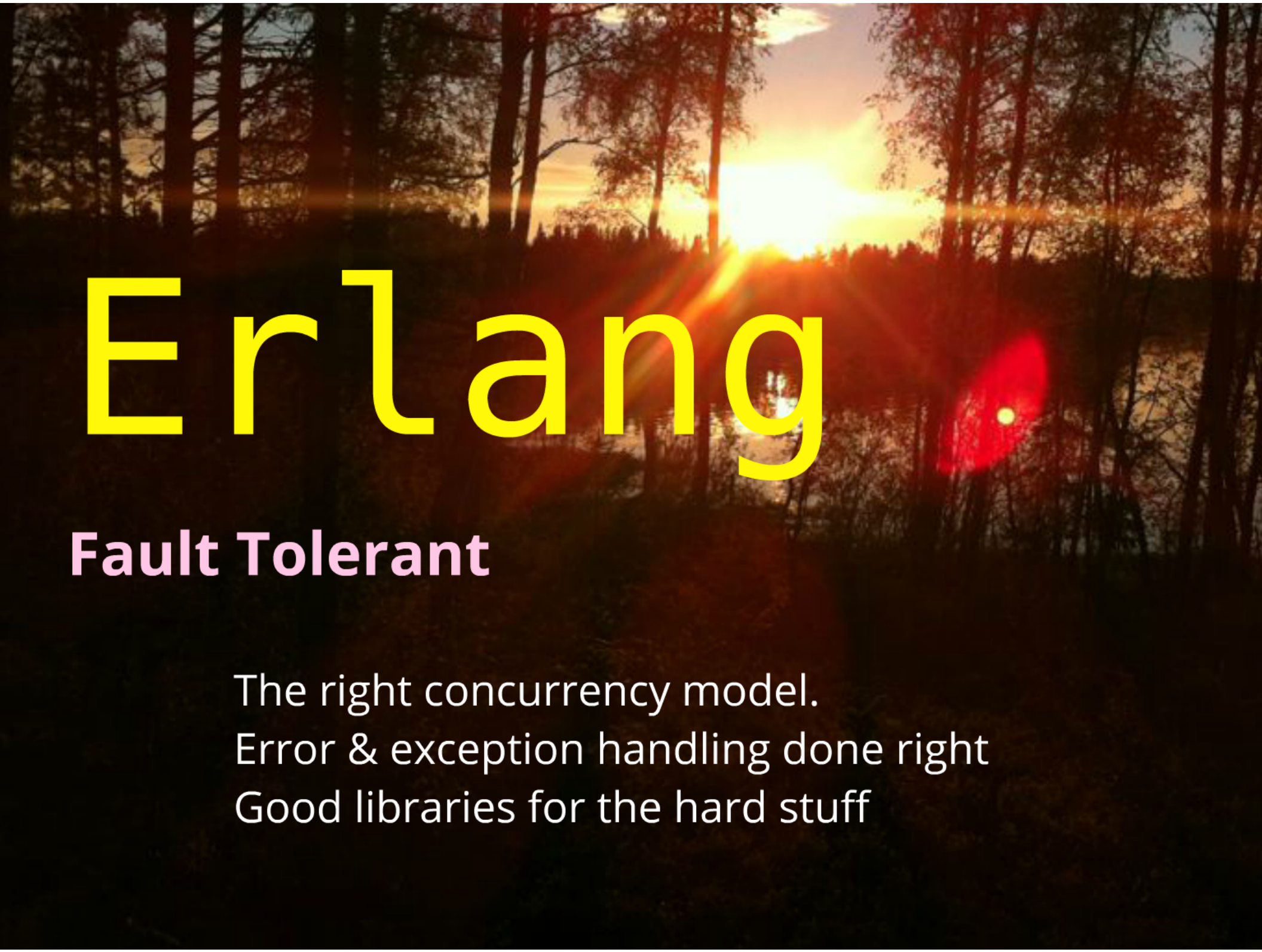
ERTS as memory areas:



Sharing




The Erlang Run-Time
System: ERTS


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

The right concurrency model

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

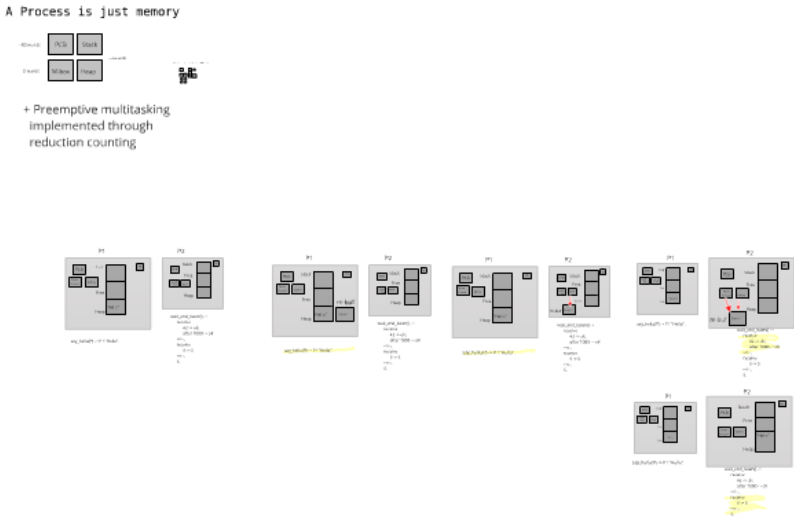
Dynamically &
Strongly
Typed

Symbolic and transparent data structures

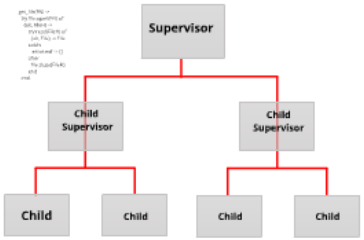
Remember
I don't have
opinions
this is all
facts.

The right concurrency model

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

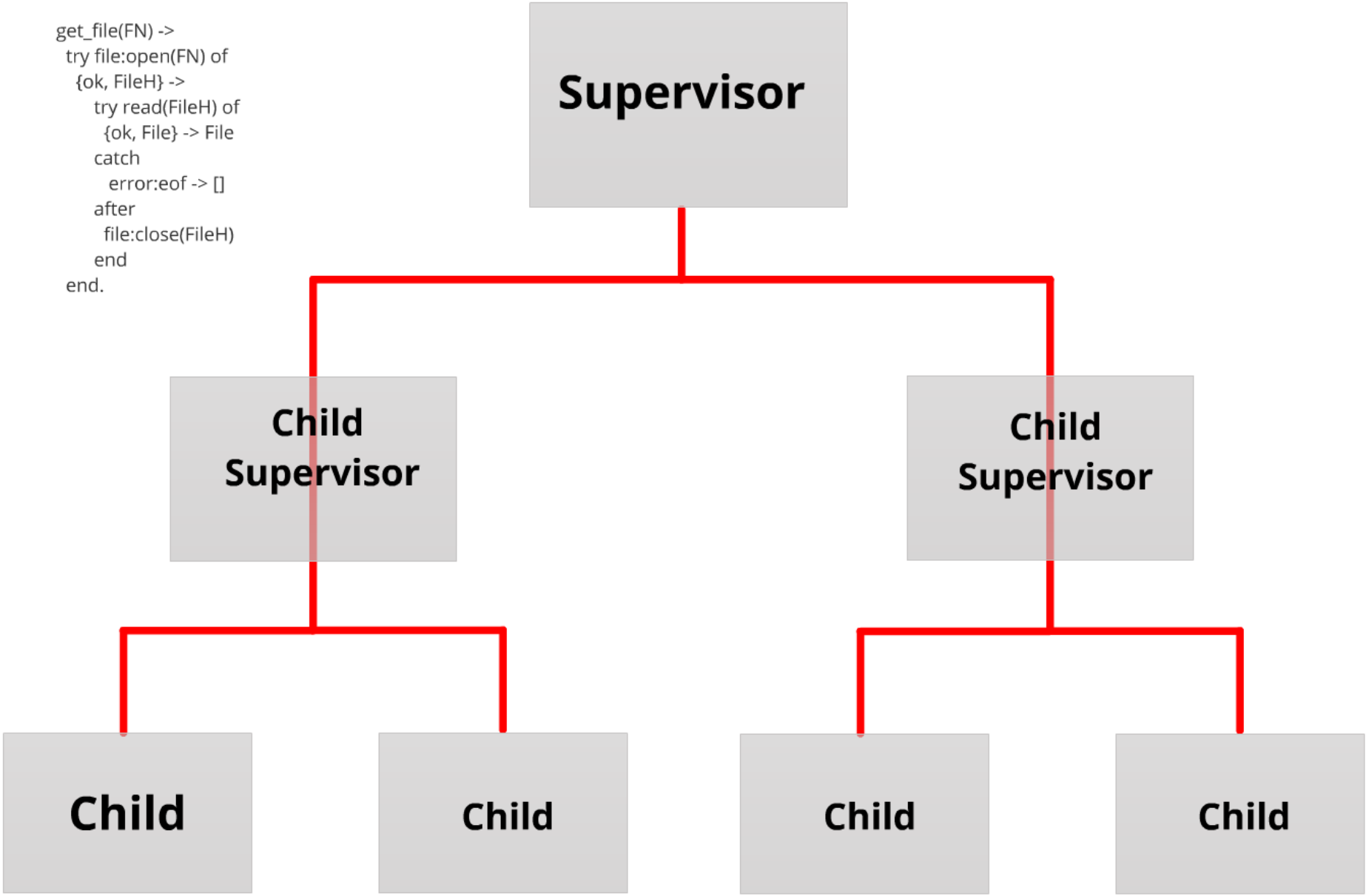


Error & exception handling done right



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.
```



Dynamically & Strongly Typed

Symbolic and transparent data structure

Enables:

Hot code loading

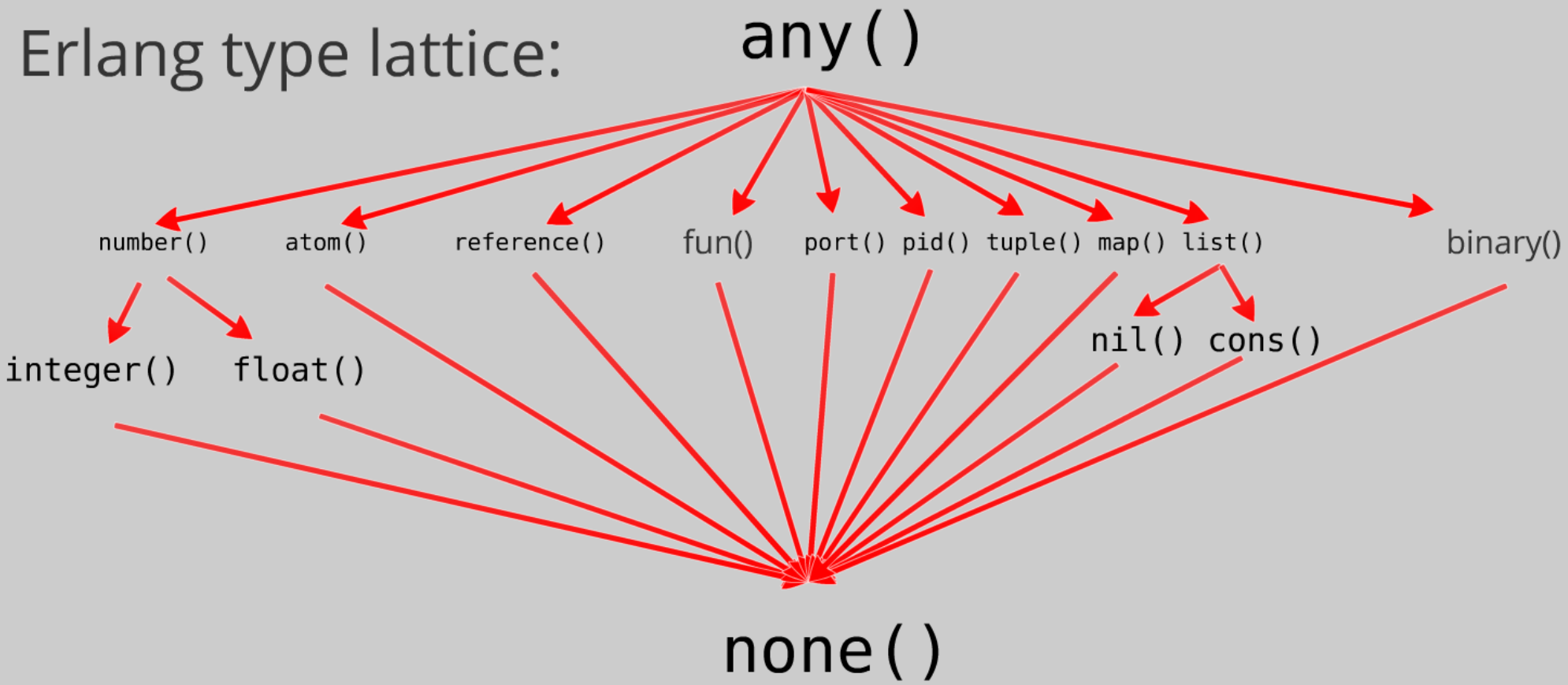
Movable heaps & stacks

Transparency of data

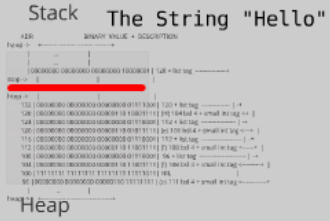
Garbage Collection

All values are tagged, some are boxed.

Erlang type lattice:



All values are tagged, some are boxed.



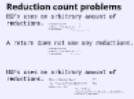
What is ERTS?

ERTS is the Erlang Runtime System.

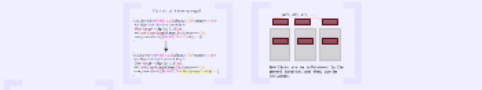
ERTS as components:

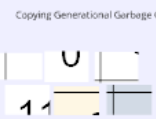
The BEAM interpreter


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

Processes

Conceptually: 4 memory areas and a pointer:

A Stack

A Heap

A Mailbox

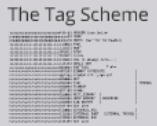
A Process Control Block

A PID

ERTS as memory areas:

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...



C-stack

Queues

Binaries

Processes

P1

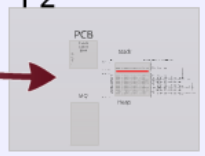
P2

P3

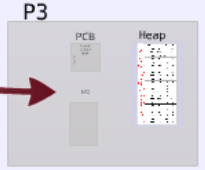
...

P99

P2



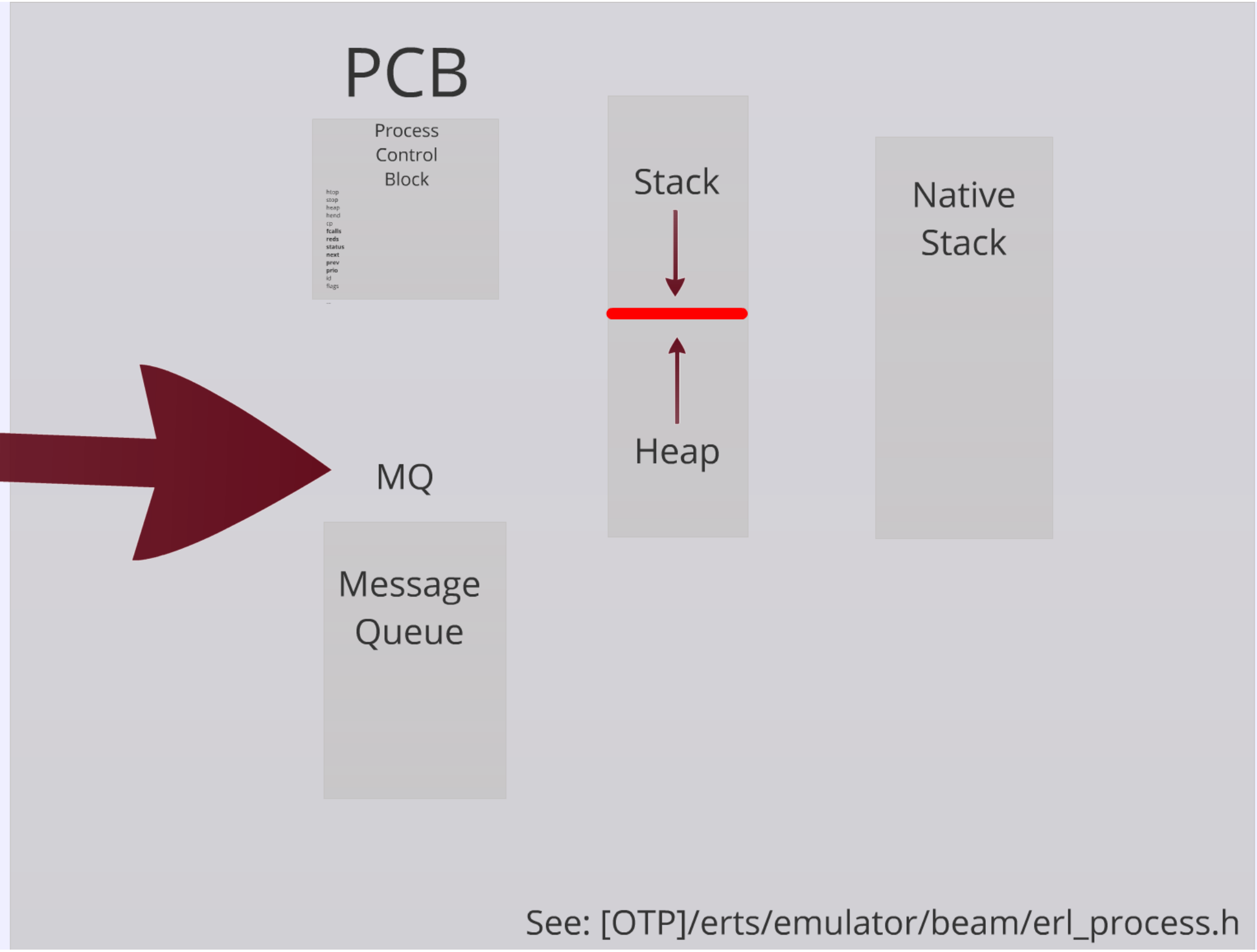
P3



...

BEAM CODE

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


PCB
Process
Control
Block
htop
stop
heap
hend
cp
fcalls
reds
status
next
prev
prio
id
flags
Stack
Heap
Native
Stack
MQ
Message
Queue
See: [OTP]/erts/emulator/beam/erl_process.h

Process Control Block

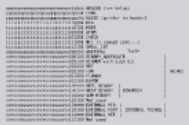
htop
stop
heap
hend
cp
fcalls
reds
status
next
prev
prio
id
flags

ERTS as meme

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...

The Tag Scheme



C-stack

Queues

Binaries

Proce

P1

P2

P3

...

P99

The Tag Scheme

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaatttt00 HEADER (see below)
ppppppppppppppppppppppppppppppp01 CONS
ppppppppppppppppppppppppppppppp10 BOXED (pointer to header)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011 PID
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111 PORT
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011 ATOM
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011 CATCH
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011 NIL (i always zero...)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111 SMALL_INT
aaaaaaaaaaaaaaaaaaaaaaaaaaa000000 ARITYVAL          Tuple
vvvvvvvvvvvvvvvvvvvvvvvvvv000100 BINARY_AGGREGATE                  |
vvvvvvvvvvvvvvvvvvvvvvvvvv001x00 BIGNUM with sign bit             |
vvvvvvvvvvvvvvvvvvvvvvvvvv010000 REF                              |
vvvvvvvvvvvvvvvvvvvvvvvvvv010100 FUN                              | THINGS
vvvvvvvvvvvvvvvvvvvvvvvvvv011000 FLONUM                           |
vvvvvvvvvvvvvvvvvvvvvvvvvv011100 EXPOR                            |
vvvvvvvvvvvvvvvvvvvvvvvvvv100000 REFC_BINARY  |                   |
vvvvvvvvvvvvvvvvvvvvvvvvvv100100 HEAP_BINARY  | BINARIES          |
vvvvvvvvvvvvvvvvvvvvvvvvvv101000 SUB_BINARY   |                   |
vvvvvvvvvvvvvvvvvvvvvvvvvv101100 Not used
vvvvvvvvvvvvvvvvvvvvvvvvvv110000 EXTERNAL_PID  |                   |
vvvvvvvvvvvvvvvvvvvvvvvvvv110100 EXTERNAL_PORT | EXTERNAL THINGS   |
vvvvvvvvvvvvvvvvvvvvvvvvvv111000 EXTERNAL_REF  |                   |
vvvvvvvvvvvvvvvvvvvvvvvvvv111100 Not used
```

An example

The string "Hello", i.e. the list

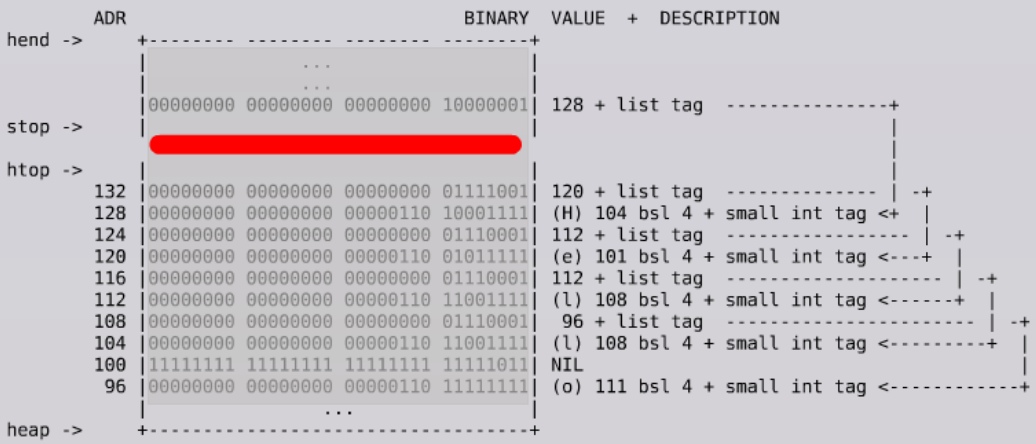
[104, 101, 108, 108, 111]

PCB

Process Control Block

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

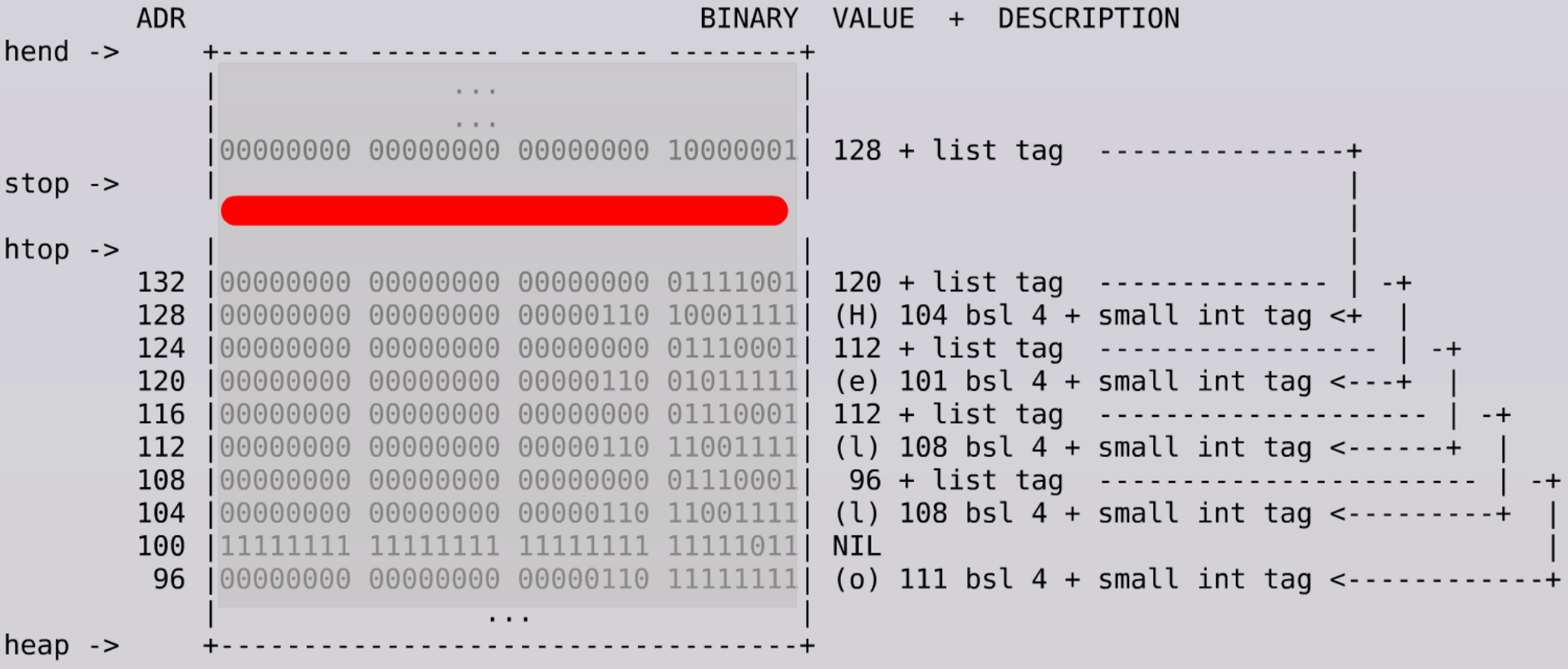
Stack



MQ

Heap

Stack

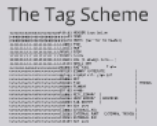

ADR
BINARY
VALUE + DESCRIPTION
hend ->
stop ->
htop ->
heap ->
00000000 00000000 00000000 10000001 128 + list tag
132 00000000 00000000 00000000 01111001 120 + list tag
128 00000000 00000000 00000110 10001111 (H) 104 bsl 4 + small int tag
124 00000000 00000000 00000000 01110001 112 + list tag
120 00000000 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

ERTS as memory areas:

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...



C-stack

Queues

Binaries

Processes

P1

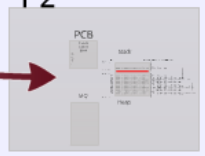
P2

P3

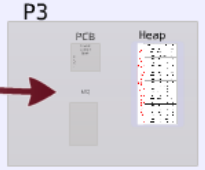
...

P99

P2



P3



...

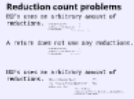
BEAM CODE

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

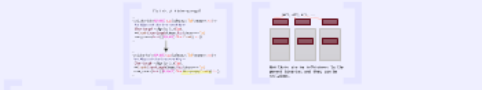
ERTS as components:

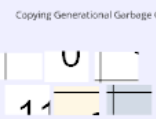
The BEAM interpreter


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

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
```

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

beam_emu.c **:

```
        OpCase(move_xx): {
0x3000:   x(Arg(1)) = x(Arg(0));
          I += 3;
          Goto(*I);
        }

        OpCase(move_yx): {
0x3200:   x(Arg(1)) = y(Arg(0));
          I += 3;
          Goto(*I);
        }

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

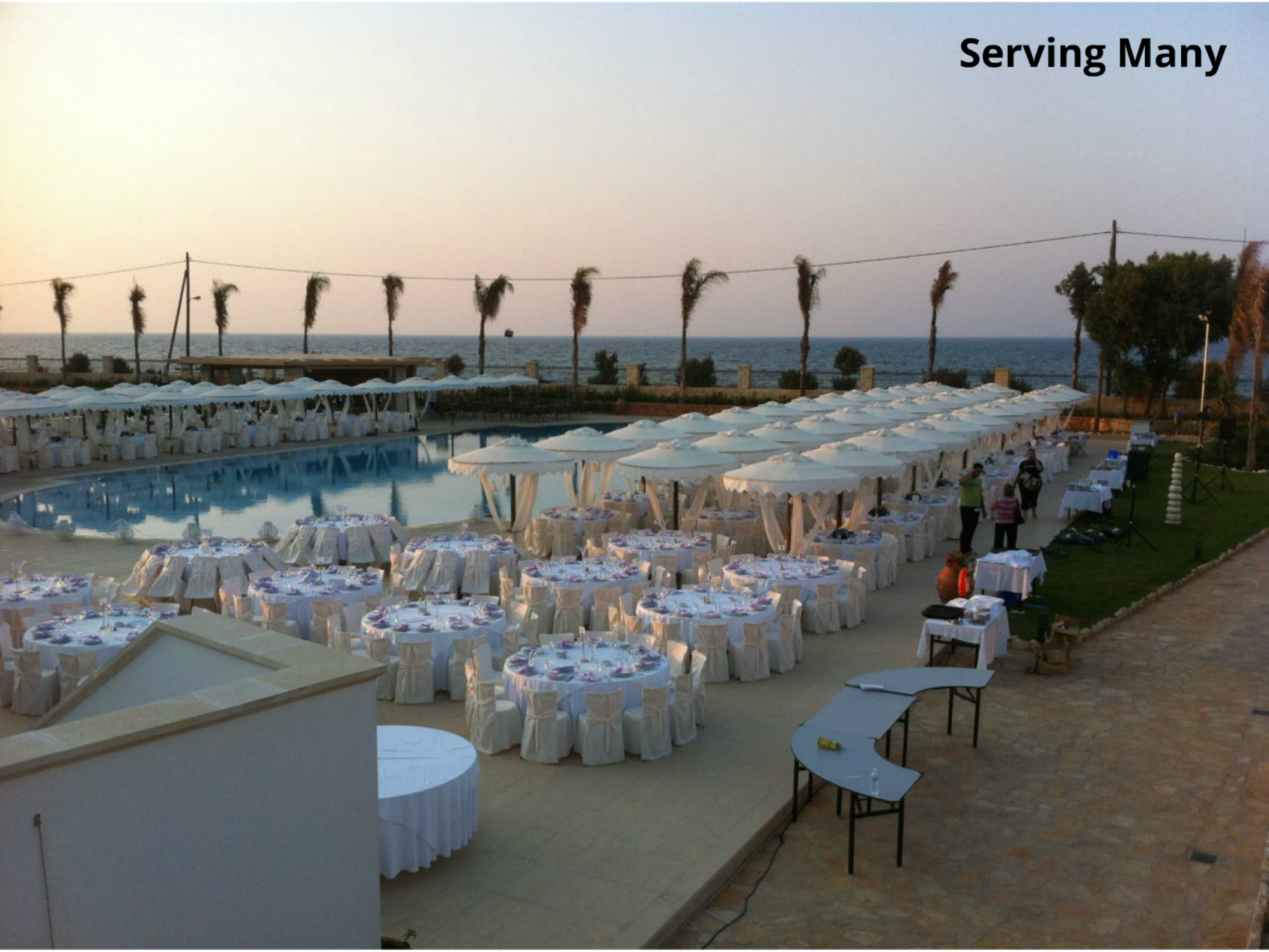
BEAM

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

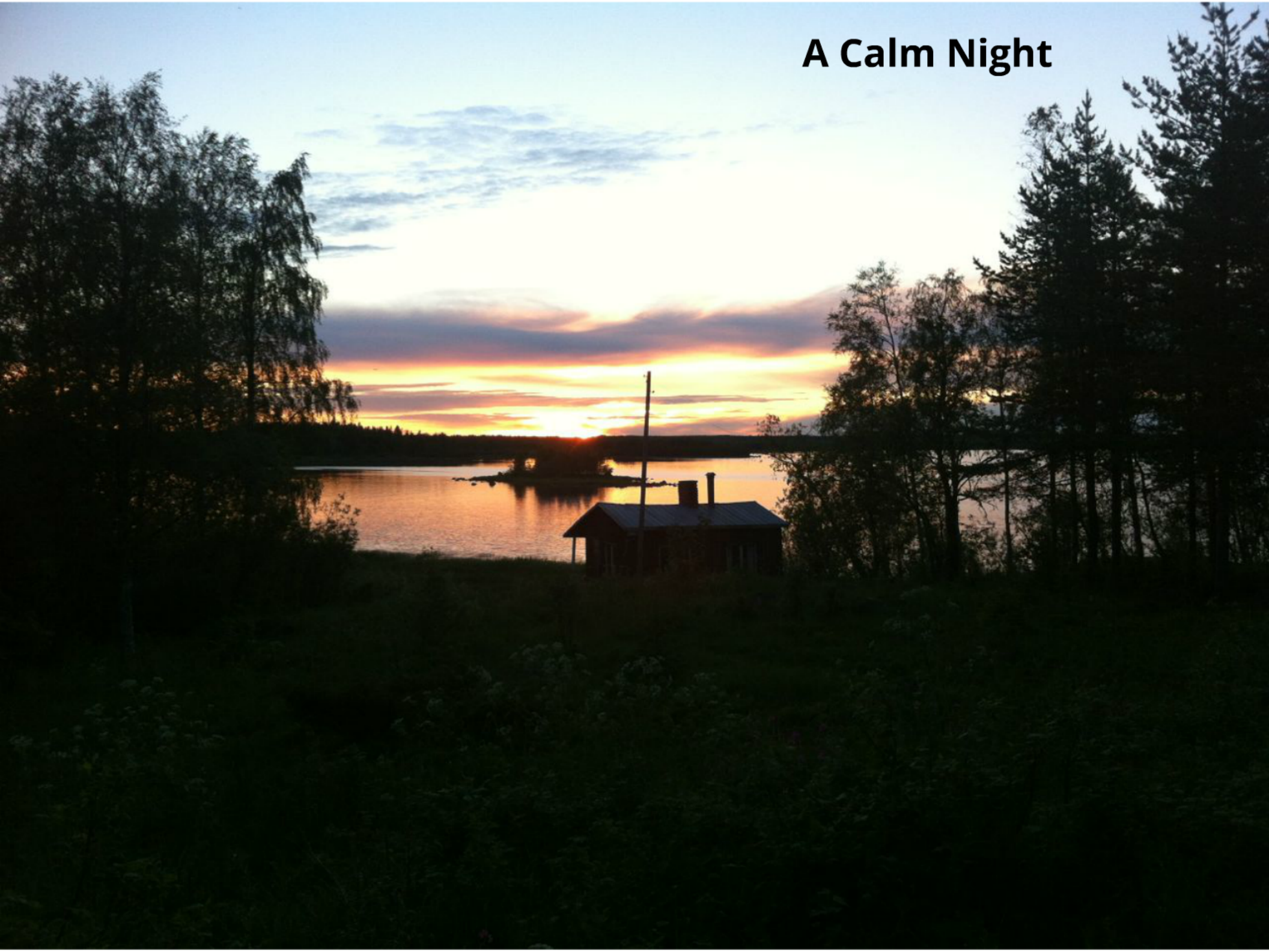
-Machine


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> c(world, 'S').

1> compile:file(world, ['S', binary]).

Beam Instructions

An Added Example

```
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: [OTP]/erts/emulator/beam/beam_emu.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

erl_process.c schedule()

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

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 scheduller work (load, free, trace, etc)
6. If function_calls > 4000 check IO, update time
7. Execute 1 to N ports for 2000 reds

Reduction count problems

`BIFs uses an arbitrary amount of reductions.`

`A return does not use any reductions.`

`NIFs uses an arbitrary amount of reductions.`

Should i be worried? Yes.
Do I need to know about this? Yes.
What can I do?

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.

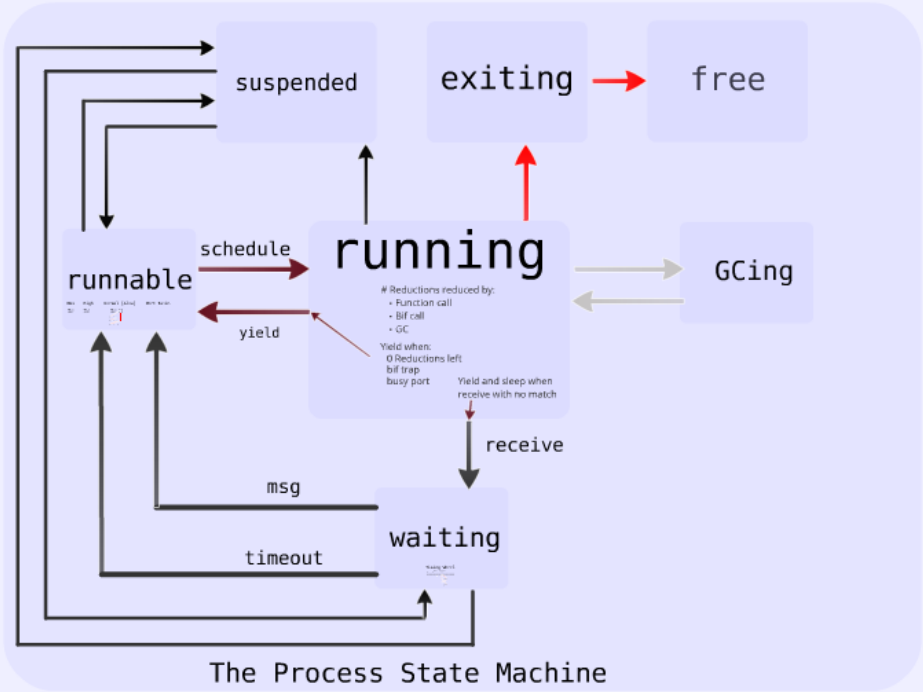
If a scheduler is overloaded procesesses are evicted to other schedulers.

If reduction counting is messed up, starvation might occur.

Use: +sfwi to wake up sleeping schedulers.

The Garbage Co

One Scheduler Per Core



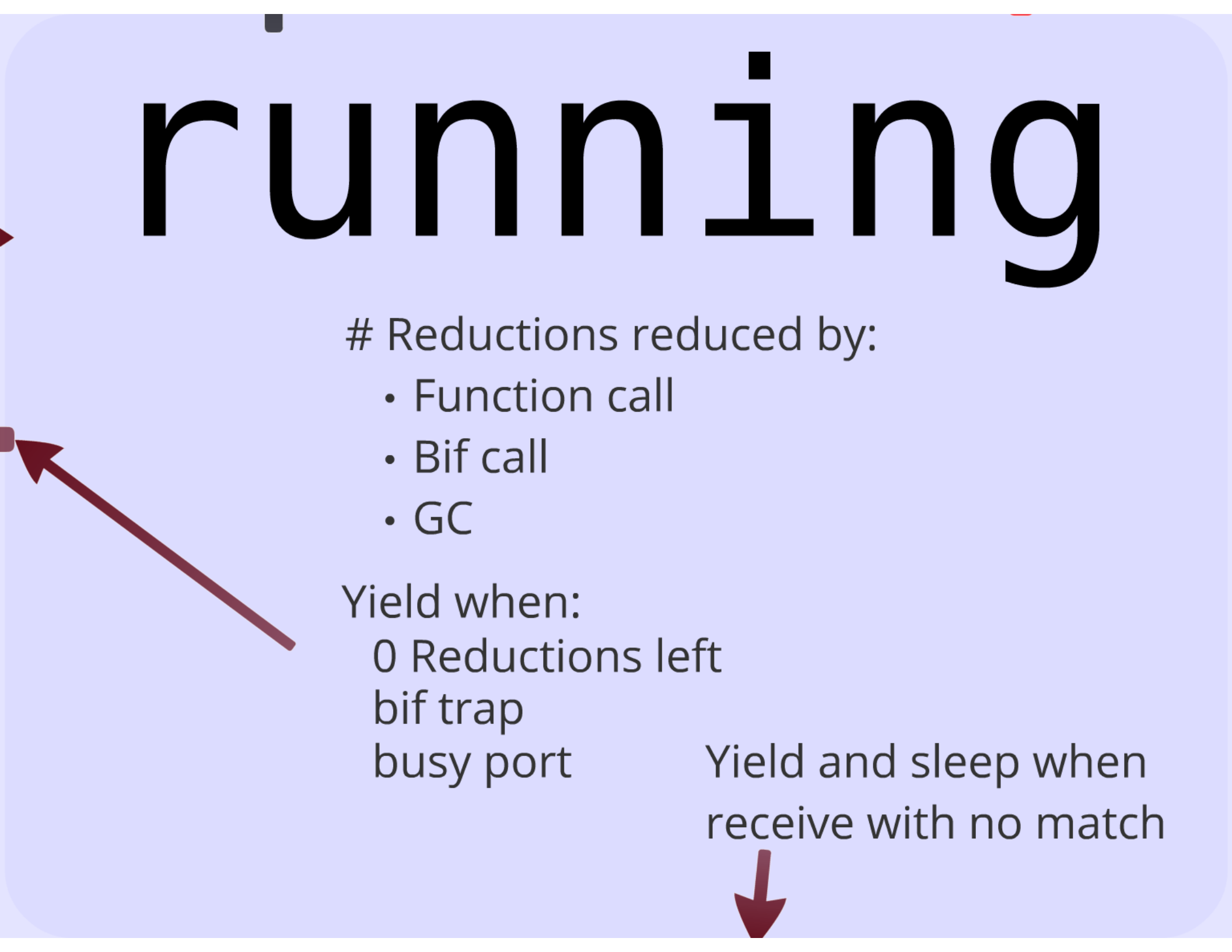
The Process State Machine

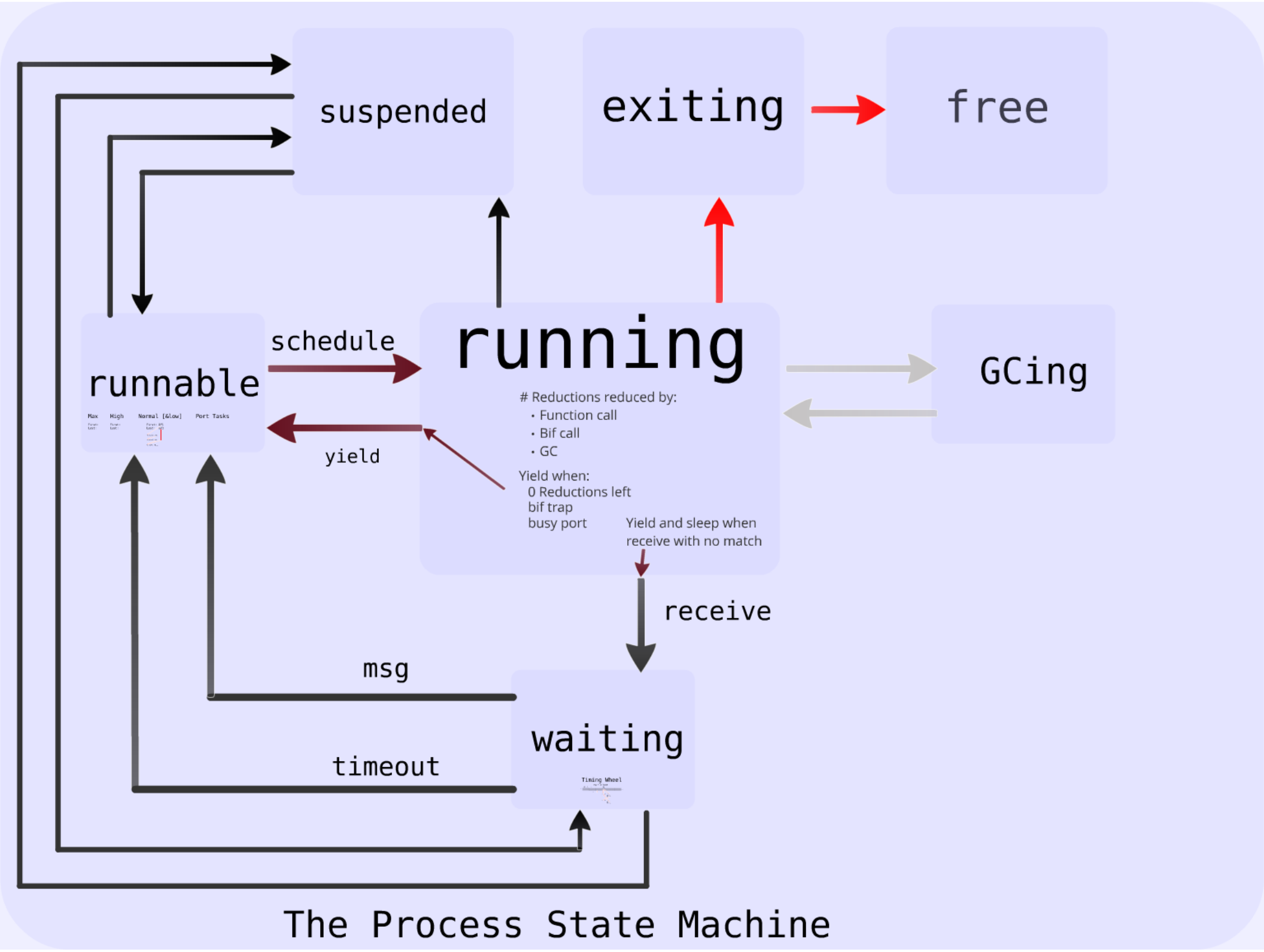
Process State
Reductions
Que Handling
Timing wheels

Possible Problems

Priority Inversion

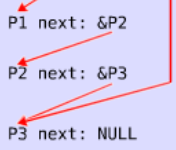
Should I be worried?	No
Do I need to know about this?	No
What can I do?	Don't mess with priorities


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

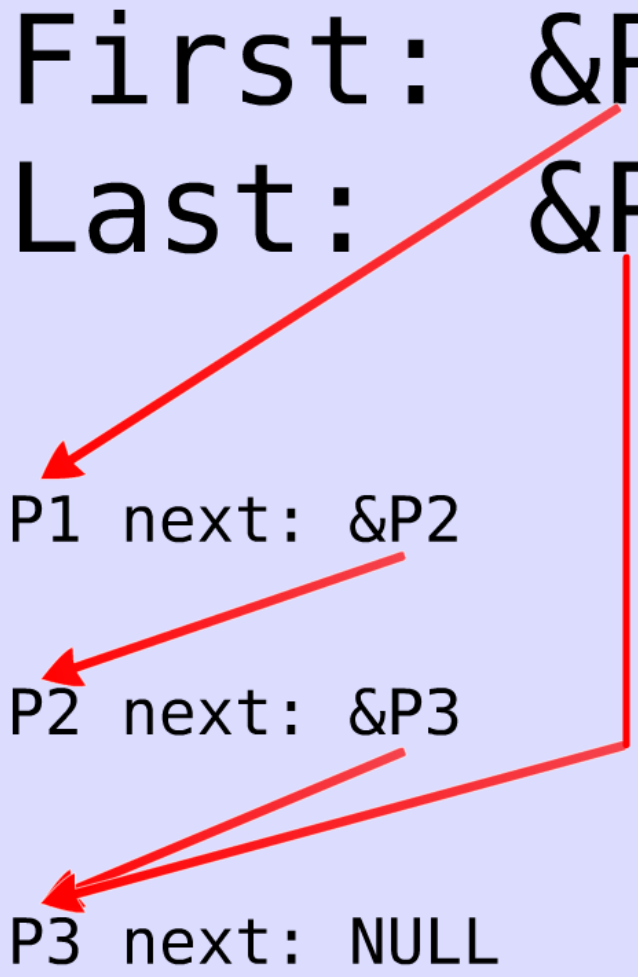

suspended
exiting
free
runnable
schedule
running
GCing
Reductions reduced by:
• Function call
• Bif call
• GC
yield
Yield when:
0 Reductions left
bif trap
busy port
Yield and sleep when
receive with no match
receive
msg
waiting
timeout
The Process State Machine

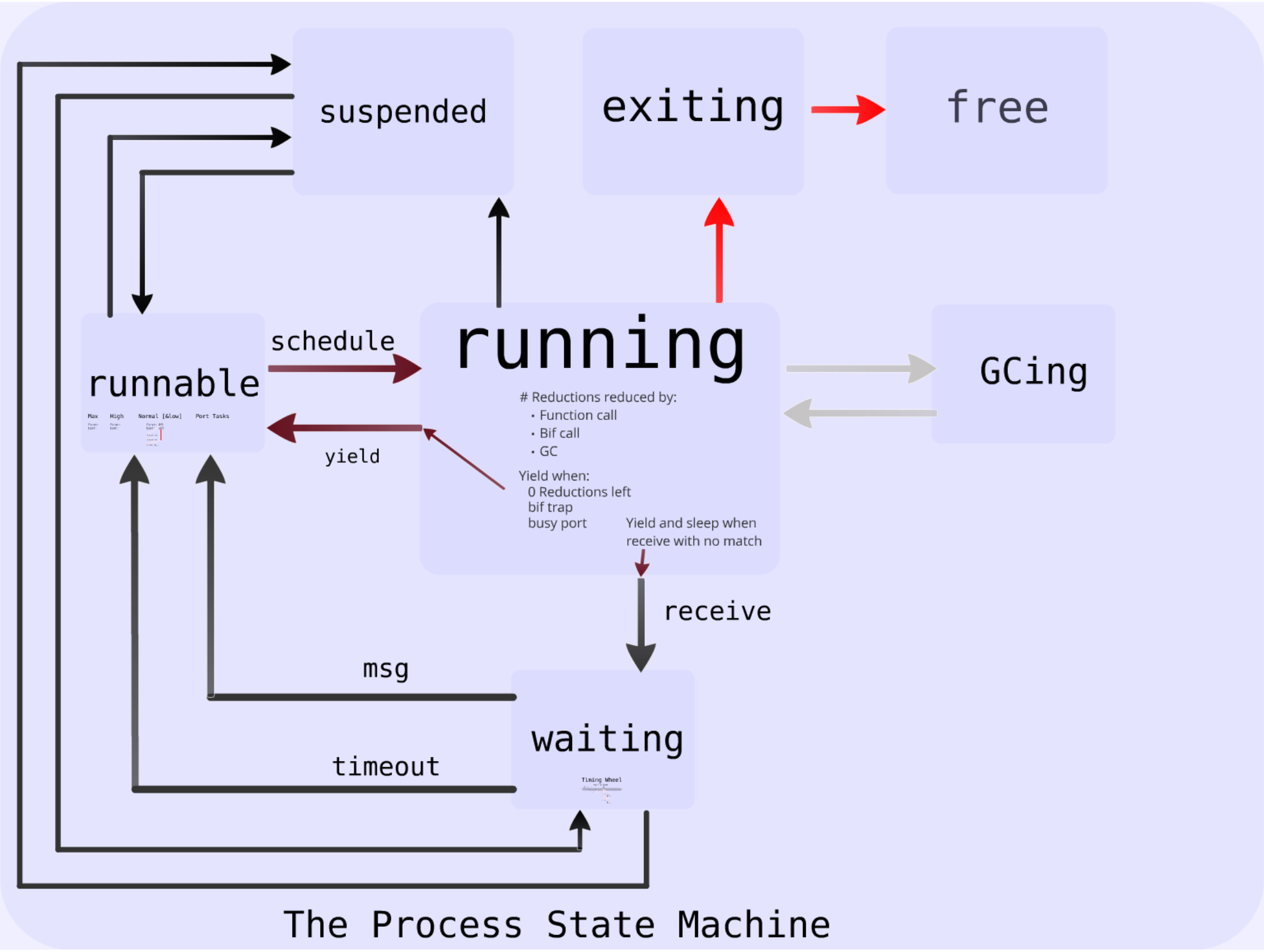
runnable

Max	High	Normal [&low]	Port Tasks
First:	First:	First: &P1	
Last:	Last:	Last: &P3	



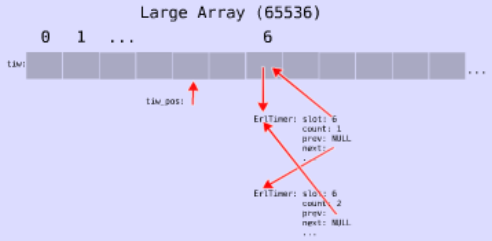
Normal [&low]



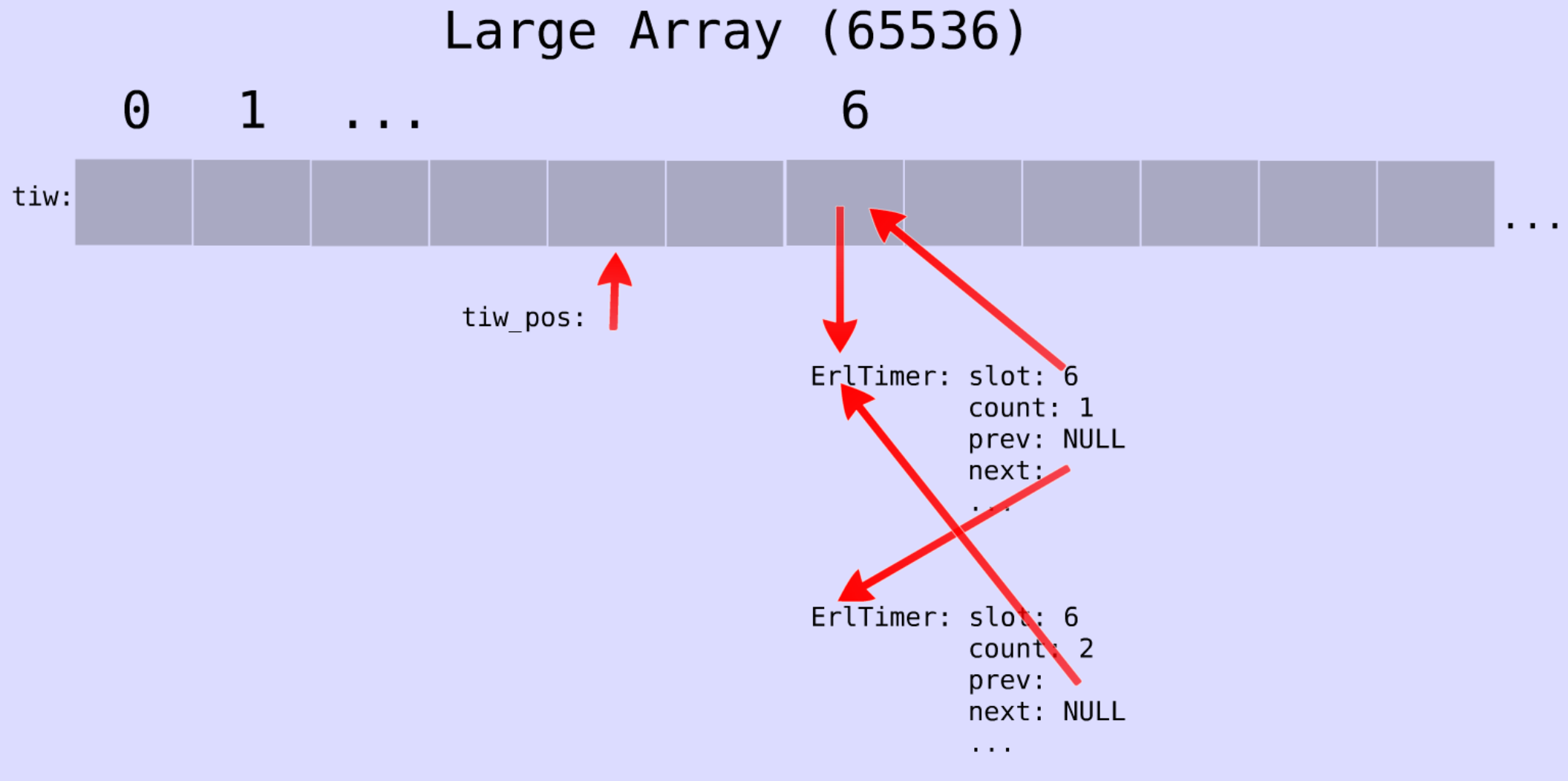

suspended
exiting
free
runnable
schedule
running
GCing
Reductions reduced by:
• Function call
• Bif call
• GC
yield
Yield when:
0 Reductions left
bif trap
busy port
Yield and sleep when
receive with no match
receive
msg
waiting
timeout
The Process State Machine

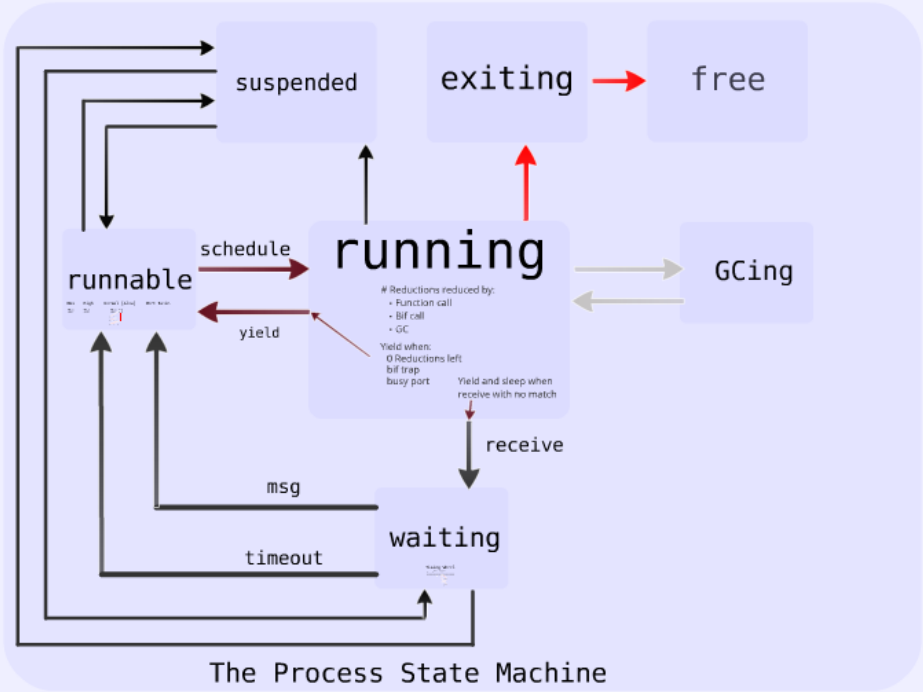
waiting

Timing Wheel



Timing Wheel





The Process State Machine

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 scheduller work (load, free, trace, etc)
6. If function_calls > 4000 check IO, update time
7. Execute 1 to N ports for 2000 reds

(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 procesesses 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".

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_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".

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? 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.

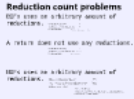
Should I be worried? Yes.
Do I need to know about this? Yes.
What can I do?

Don't use NIFs ;)
Make sure your NIFs are yielding and using reductions.
Wait for "dirty schedulers".

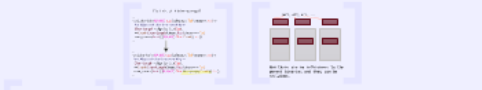
ERTS as components:

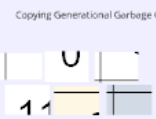
The BEAM interpreter


The Scheduler

The Garbage Collector

Processes

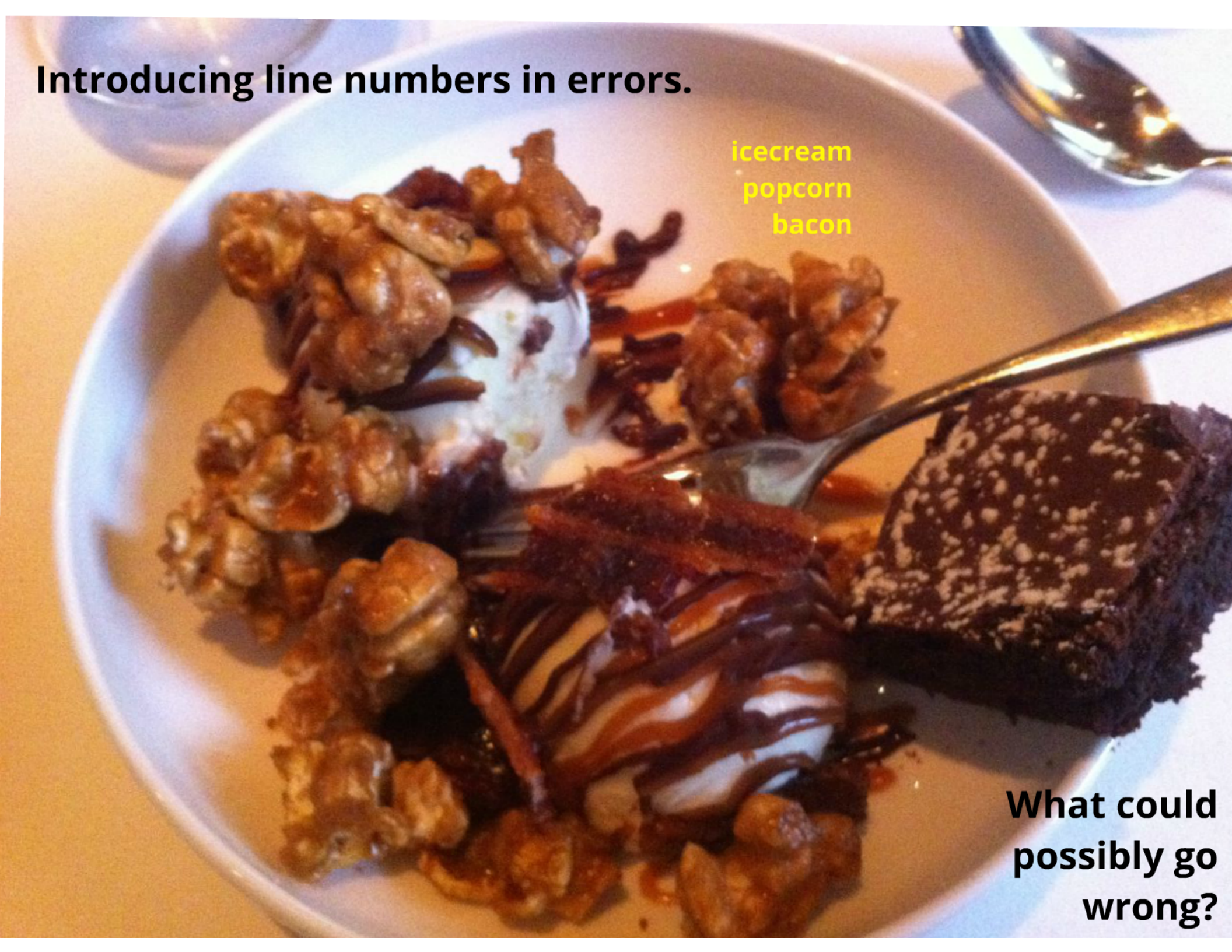
Conceptually: 4 memory areas and a pointer:

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

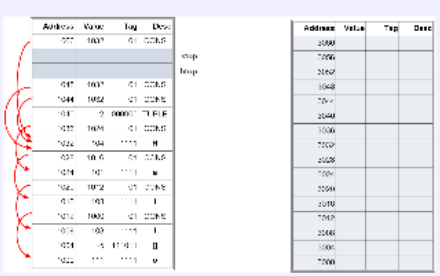
HiPE

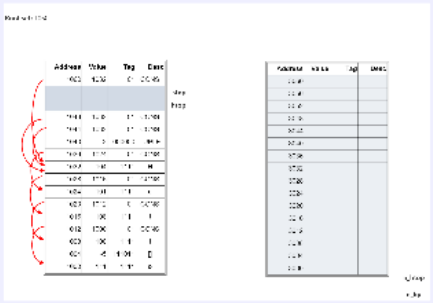
I/O

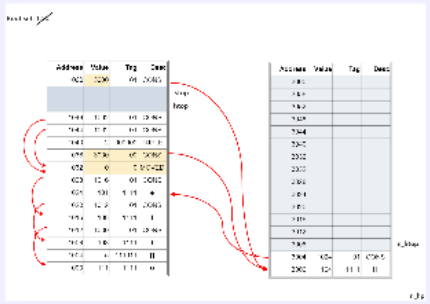

Another War Story

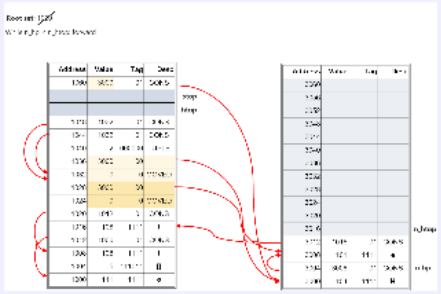

Introducing line numbers in errors.
icecream
popcorn
bacon
What could possibly go wrong?

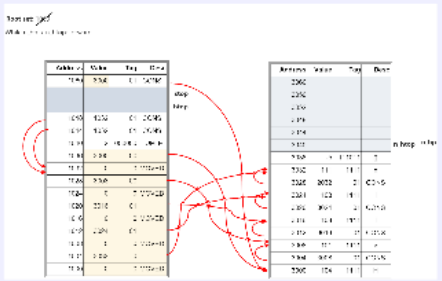
Copying Generational Garbage Collector

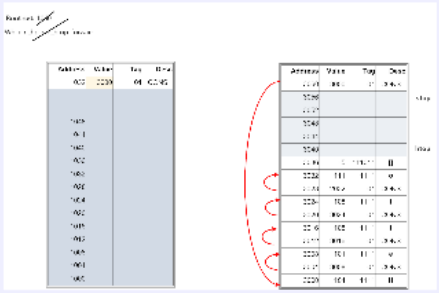


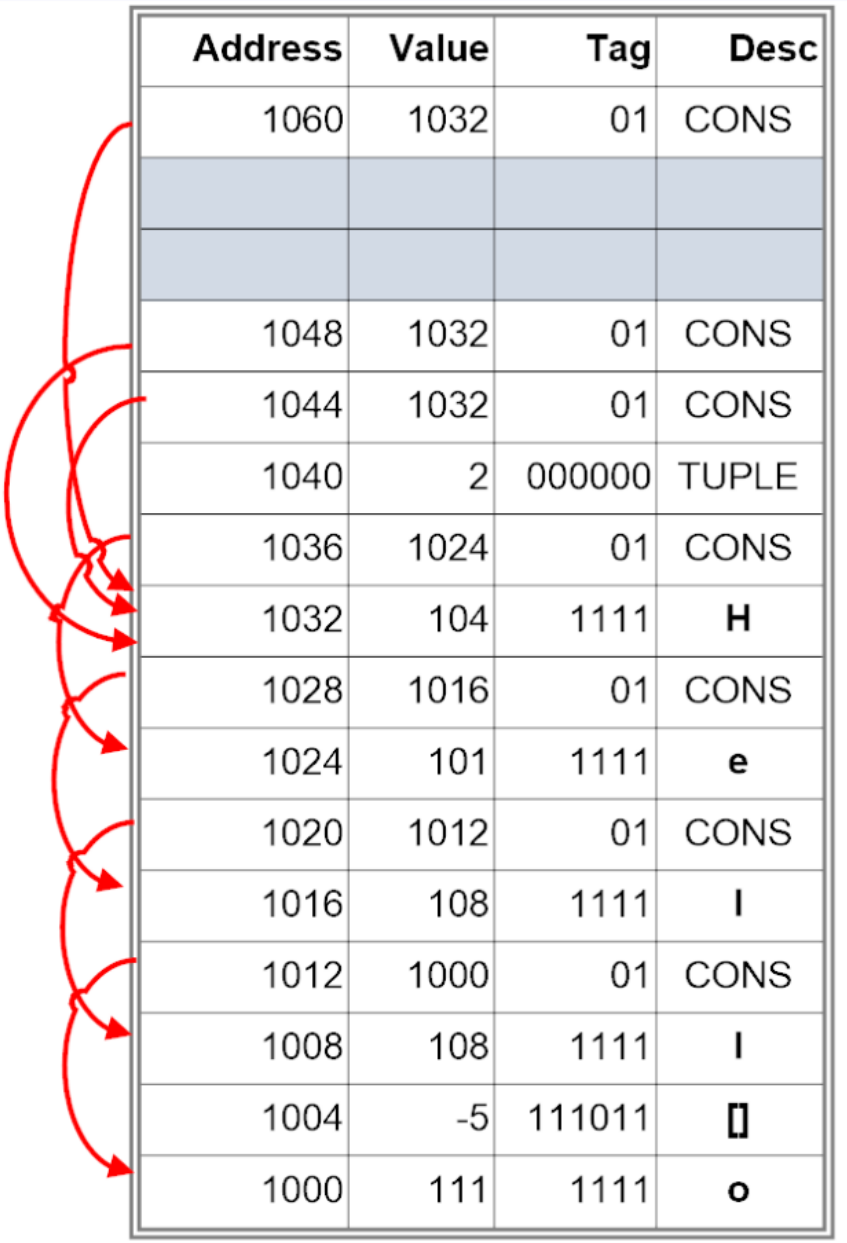












Address	Value	Tag	Desc
1060	1032	01	CONS
1048	1032	01	CONS
1044	1032	01	CONS
1040	2	000000	TUPLE
1036	1024	01	CONS
1032	104	1111	**H**
1028	1016	01	CONS
1024	101	1111	**e**
1020	1012	01	CONS
1016	108	1111	**l**
1012	1000	01	CONS
1008	108	1111	**l**
1004	-5	111011	**[]**
1000	111	1111	**o**

stop

htop

Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036			
3032			
3028			
3024			
3020			
3016			
3012			
3008			
3004			
3000			

Root set: 1060

Address	Value	Tag	Desc
1060	1032	01	CONS
1048	1032	01	CONS
1044	1032	01	CONS
1040	2	000000	TUPLE
1036	1024	01	CONS
1032	104	1111	H
1028	1016	01	CONS
1024	101	1111	e
1020	1012	01	CONS
1016	108	1111	l
1012	1000	01	CONS
1008	108	1111	l
1004	-5	111011	[]
1000	111	1111	o

stop

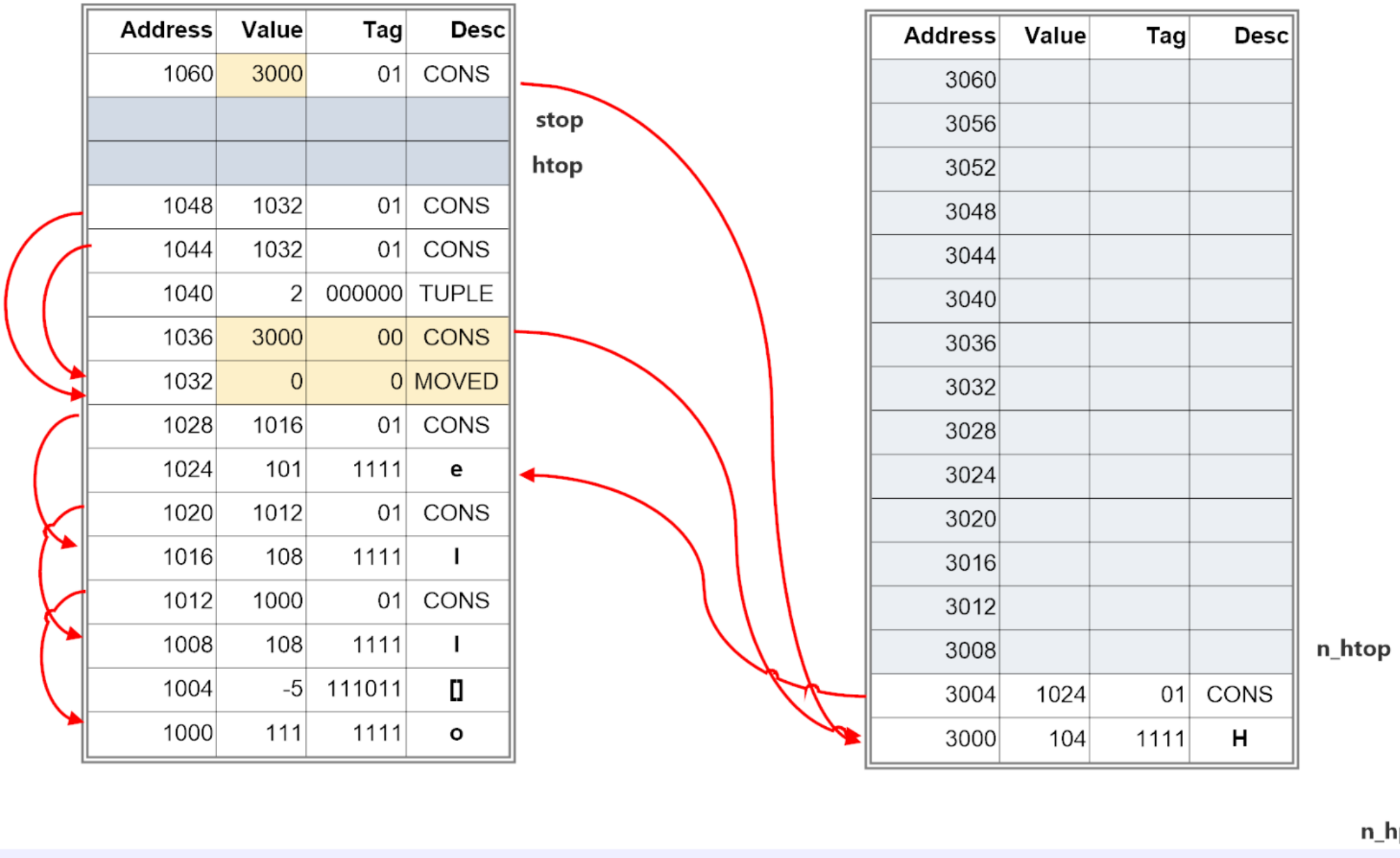
htop

Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036			
3032			
3028			
3024			
3020			
3016			
3012			
3008			
3004			
3000			

n_htop

n_hp

Root set: ~~1060~~

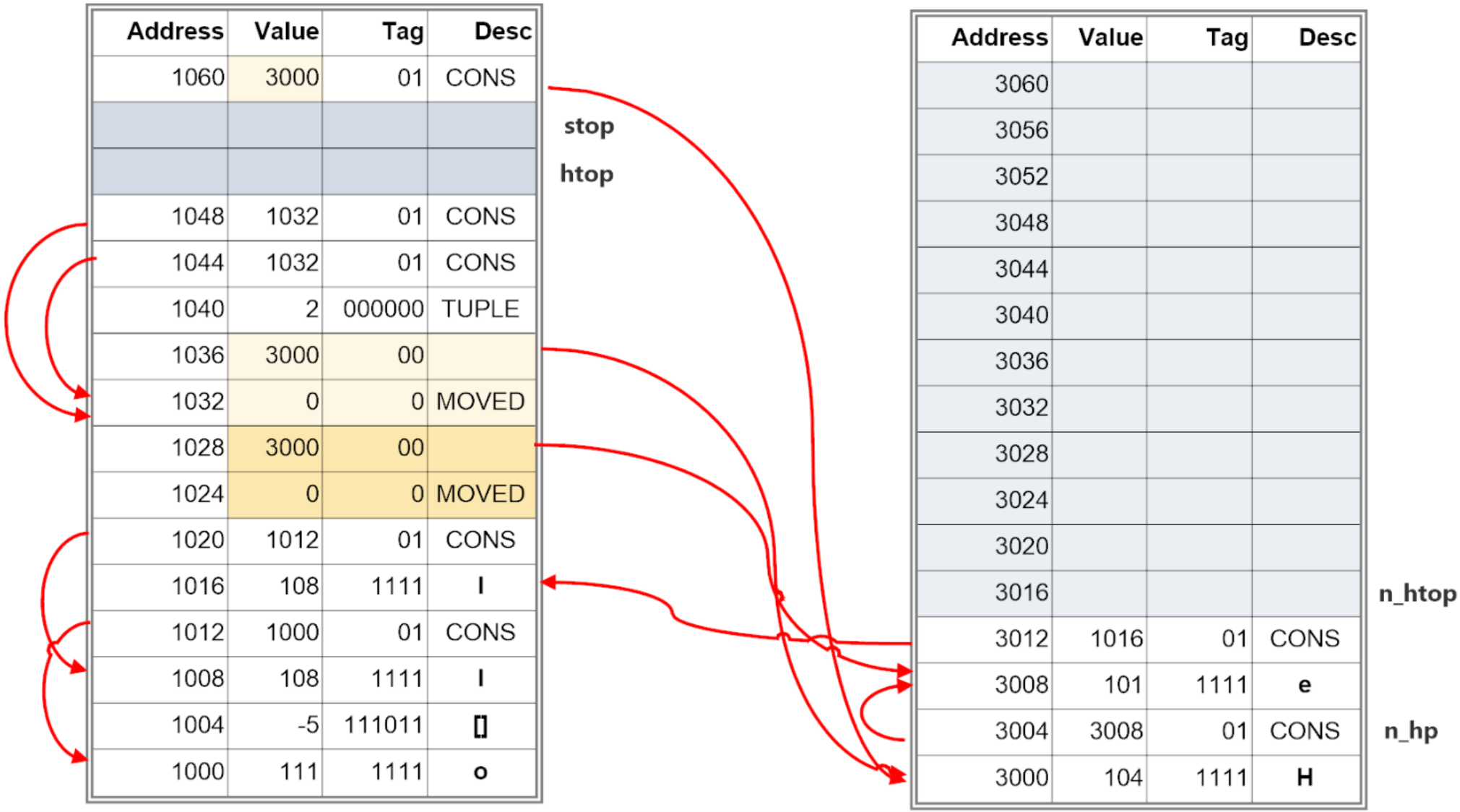


Address	Value	Tag	Desc
1060	3000	01	CONS
1048	1032	01	CONS
1044	1032	01	CONS
1040	2	000000	TUPLE
1036	3000	00	CONS
1032	0	0	MOVED
1028	1016	01	CONS
1024	101	1111	e
1020	1012	01	CONS
1016	108	1111	l
1012	1000	01	CONS
1008	108	1111	l
1004	-5	111011	[]
1000	111	1111	o

Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036			
3032			
3028			
3024			
3020			
3016			
3012			
3008			
3004	1024	01	CONS
3000	104	1111	H

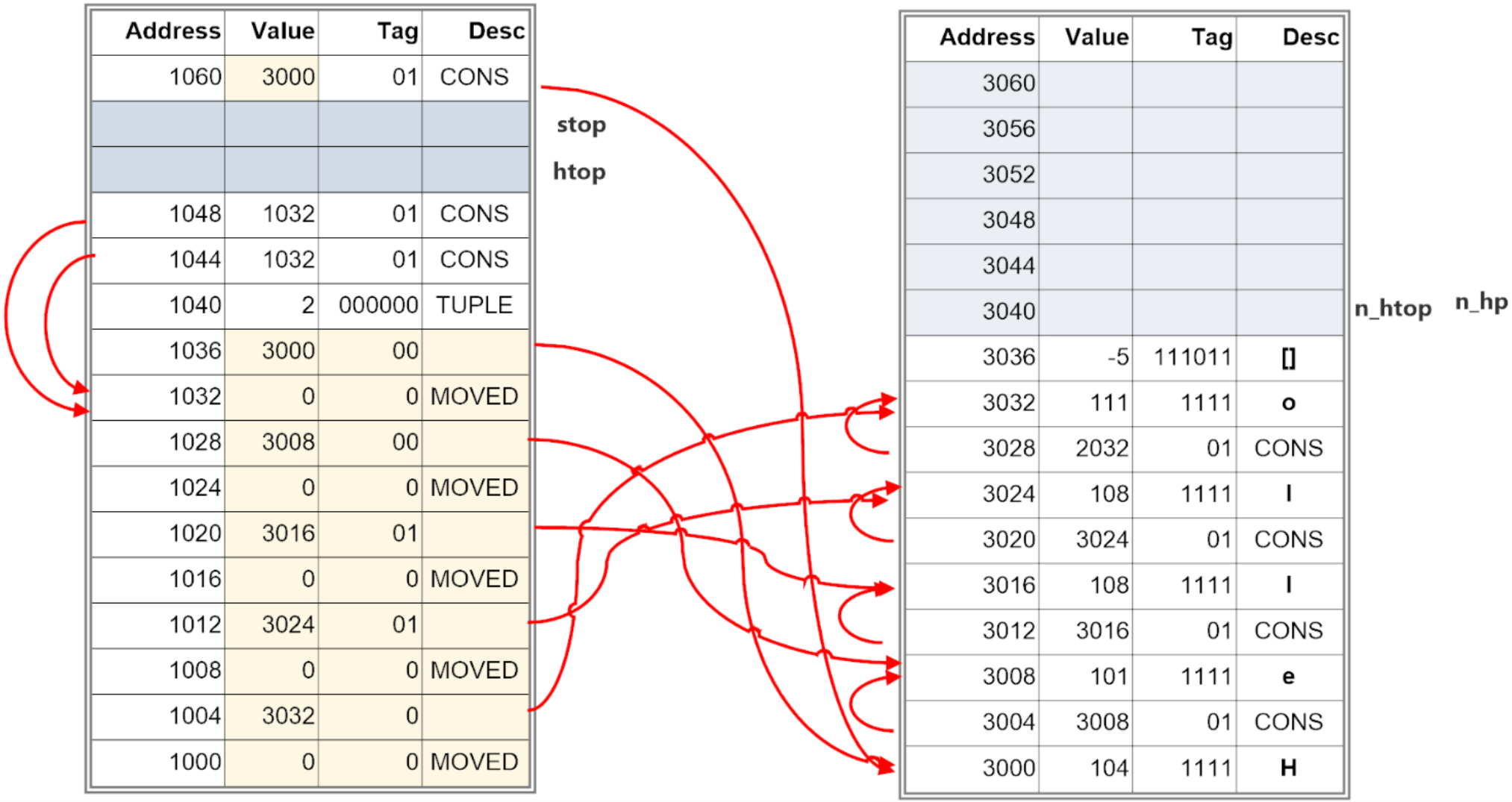
Root set: 1060

While n_hp < n_htop: forward



Root set: 1060

While n_hp < n_htop: forward



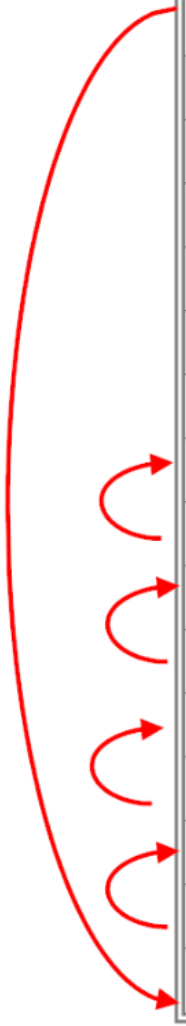
Address	Value	Tag	Desc
1060	3000	01	CONS
1048	1032	01	CONS
1044	1032	01	CONS
1040	2	000000	TUPLE
1036	3000	00	
1032	0	0	MOVED
1028	3008	00	
1024	0	0	MOVED
1020	3016	01	
1016	0	0	MOVED
1012	3024	01	
1008	0	0	MOVED
1004	3032	0	
1000	0	0	MOVED

Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036	-5	111011	[]
3032	111	1111	o
3028	2032	01	CONS
3024	108	1111	l
3020	3024	01	CONS
3016	108	1111	l
3012	3016	01	CONS
3008	101	1111	e
3004	3008	01	CONS
3000	104	1111	H

Root set: 1060

While n_hp < n_htop: forward

Address	Value	Tag	Desc
1032	3000	01	CONS
1048			
1044			
1040			
1036			
1032			
1028			
1024			
1020			
1016			
1012			
1008			
1004			
1000			



Address	Value	Tag	Desc	
3060	3000	01	CONS	
3056				stop
3052				
3048				
3044				
3040				htop
3036	-5	111011	[]	
3032	111	1111	o	
3028	2032	01	CONS	
3024	108	1111	l	
3020	3024	01	CONS	
3016	108	1111	l	
3012	3016	01	CONS	
3008	101	1111	e	
3004	3008	01	CONS	
3000	104	1111	H	

Copying Generational Garbage Collector

Advantages with 1 heap/process:

+ Free reclamation when a process dies
+ Small root set
+ Sort of "Incremental"
+ Improved cache locallity
+ Cheap stack/heap test
+ Smal 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
What is Erlang?
Erlang the language
Open Source
Mnesia
Fault tolerant
Erlang
1974
NATL LEAGUE CHAMPIONS
KIRIN BEER

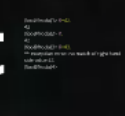
Erlang the language

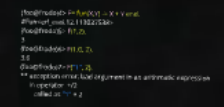
```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional

Single-assignment

Dynamically typed





Concurrent

Distributed

Message passing

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").
```

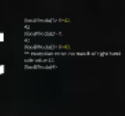
Erlang the language

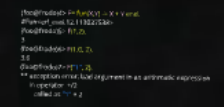
```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional

Single-assignment

Dynamically typed





Concurrent

Distributed

Message passing

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
```

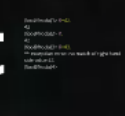
Erlang the language

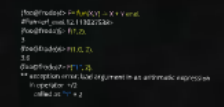
```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional

Single-assignment

Dynamically typed





Concurrent

Distributed

Message passing

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>
```

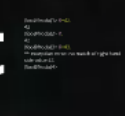
Erlang the language

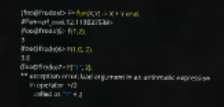
```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional

Single-assignment

Dynamically typed





Concurrent

Distributed

Message passing

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
```

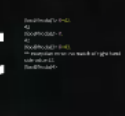
Erlang the language

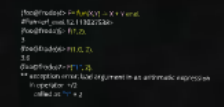
```
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World!\n").
```

Functional

Single-assignment

Dynamically typed





Concurrent

Distributed

Message passing

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}.
```

Mnesia

Hot code loading

```
lm() -> [c:l(M) || M <- mm()].
mm() -> modified_modules().
```

Hof on DB

QLC

Fault tolerant

Supervisor trees

Demo

Demo3

Hof map/fold

Genserver

Atoms/symbolic programming

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
```

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