Erlang Engine Tuning:
Part 1
Know Your Engine

What is ERTS?

ERTS is the Erlang Runtime System.

ERTS as source code:

See: [OTP]/erts/

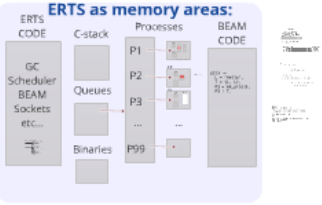
emulator/
beam/
hipe/
etc/

ERTS as components:

The BEAM interpreter
The Scheduler
The Garbage Collector

Processes
HiPE
I/O

ERTS as memory areas:



Hibernation:
Do a GC to a temp area.
Check size
Allocate a minimal mem area
move live data

Erlang Engine Tuning:
Part 1
Know Your Engine

Erik Stenman

Happi

- Programming since 1980
- Erlang since 1994
- First native code compiler for Erlang
- HiPE
- Project Manager for Scala 1.0
- 2005-2010 CTO @ Klarna
- Chief Scientist @ Klarna
- Writing a book about ERTS

What is ERTS?

ERTS is the Erlang Runtime System.

ERTS as source code:

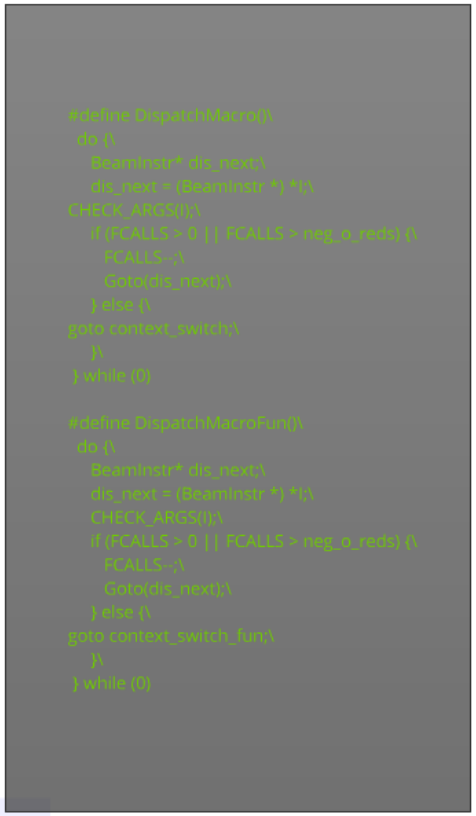
See: [OTP]/erts/

emulator/

beam/

hipe/

etc/



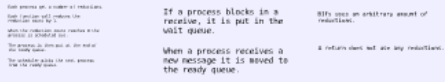
```
#define DispatchMacro()\
 do {\
    BeamInstr* dis_next;\
    dis_next = (BeamInstr *) *I;\
CHECK_ARGS(I);\
    if (FCALLS > 0 || FCALLS > neg_o_reds) {\
      FCALLS--;\
      Goto(dis_next);\
    } else {\
goto context_switch;\
    }\
 } while (0)

#define DispatchMacroFun()\
 do {\
    BeamInstr* dis_next;\
    dis_next = (BeamInstr *) *I;\
    CHECK_ARGS(I);\
    if (FCALLS > 0 || FCALLS > neg_o_reds) {\
      FCALLS--;\
      Goto(dis_next);\
    } else {\
goto context_switch_fun;\
    }\
 } while (0)
```

ERTS as components:

The BEAM interpreter

The Scheduler



The Garbage Collector

Processes

Conceptually: 4 mem areas and a pointer:

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

HiPE

I/O

Processes

Conceptually: 4 mem 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

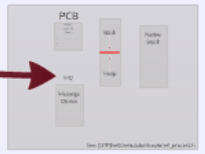
P1

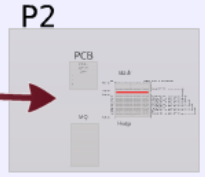
P2

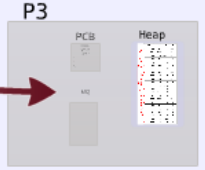
P3

...

P99



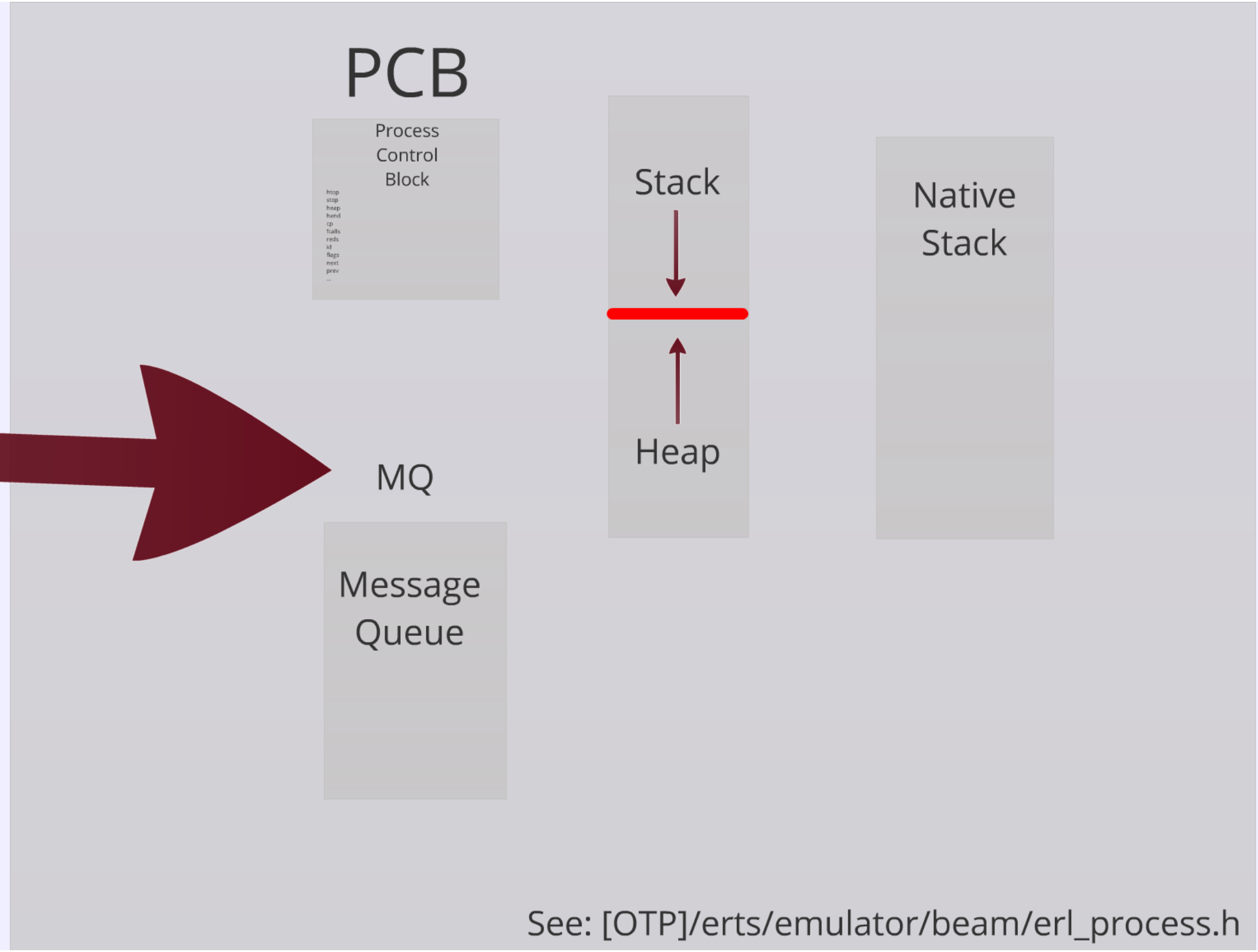




...

BEAM CODE

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

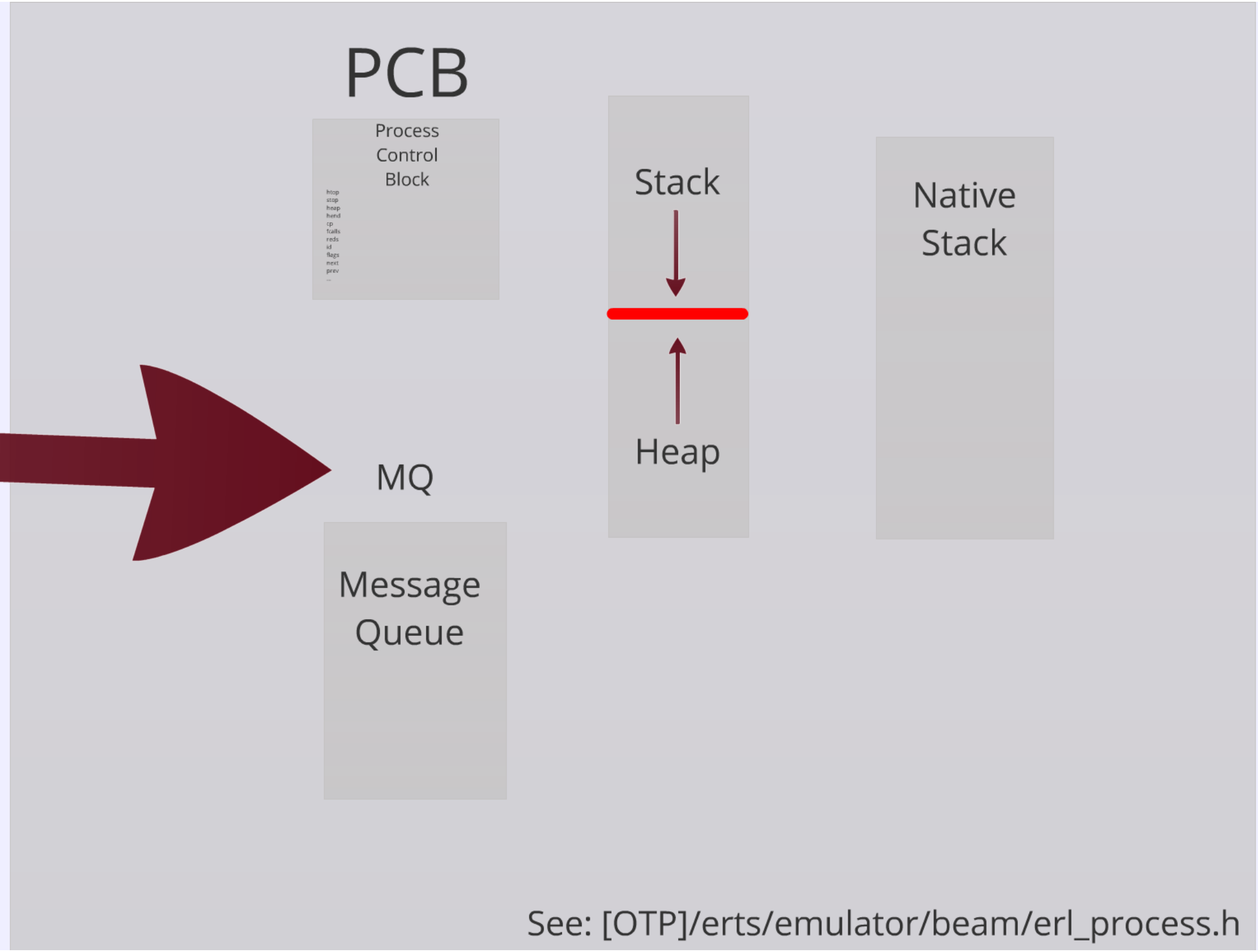

PCB
Process
Control
Block
htop
stop
heap
hend
cp
fcalls
reds
id
flags
next
prev
...
Stack
Heap
Native
Stack
MQ
Message
Queue
See: [OTP]/erts/emulator/beam/erl_process.h

Process Control Block

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

BIO

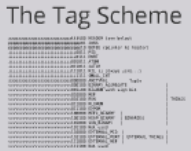
htop
stop
heap
hend
cp
fcalls
reds
id
flags
next
prev
...


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

ERTS as meme

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...



C-stack

Queues

Binaries

Proce

P1

P2

P3

...

P99

GC

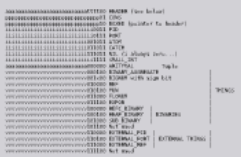
Scheduler

BEAM

Sockets

etc...

The Tag Scheme



Queues

Binaries

The Tag Scheme

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaatttt00 HEADER (see below)
pppppppppppppppppppppppppppppp01 CONS
pppppppppppppppppppppppppppppp10 BOXED (pointer to header)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011 PID
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111 PORT
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011 ATOM
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011 CATCH
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011 NIL (i always zero...)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111 SMALL_INT
aaaaaaaaaaaaaaaaaaaaaaaaaa000000 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
```

The Tag Scher

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaatttt00 HEADER (see below)
pppppppppppppppppppppppppppppp01 CONS
pppppppppppppppppppppppppppppp10 BOXED (pointer to header)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011 PID
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111 PORT
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011 ATOM
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011 CATCH
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011 NIL (i always zero...)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111 SMALL_INT
aaaaaaaaaaaaaaaaaaaaaaaaaaaa000000 ARITYVAL          Tuple
vvvvvvvvvvvvvvvvvvvvvvvvvv000100 BINARY_AGGREGATE
vvvvvvvvvvvvvvvvvvvvvvvvvv001x00 BIGNUM with sign bit
vvvvvvvvvvvvvvvvvvvvvvvvvv010000 REF
```

The Tag Scher

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaatttt00 HEADER (see below)
ppppppppppppppppppppppppppppppppp01 CONS
ppppppppppppppppppppppppppppppppp10 BOXED (pointer to header)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011 PID
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111 PORT
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011 ATOM
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011 CATCH
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011 NIL (i always zero...)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111 SMALL_INT
aaaaaaaaaaaaaaaaaaaaaaaaaa000000 ARITYVAL          Tuple
vvvvvvvvvvvvvvvvvvvvvvvvvv000100 BINARY_AGGREGATE
vvvvvvvvvvvvvvvvvvvvvvvvvv001x00 BIGNUM with sign bit
vvvvvvvvvvvvvvvvvvvvvvvvvv010000 REF
vvvvvvvvvvvvvvvvvvvvvvvvvv010100 FUN
vvvvvvvvvvvvvvvvvvvvvvvvvv011000 FLONUM
vvvvvvvvvvvvvvvvvvvvvvvvvv011100 EXPOR
vvvvvvvvvvvvvvvvvvvvvvvvvv100000 REFC_BINARY  |
vvvvvvvvvvvvvvvvvvvvvvvvvv100100 HEAP_BINARY  | BINARIES
```

```
pppppppppppppppppppppppppppppp01 CONS
pppppppppppppppppppppppppppppp10 BOXED (pointer to header)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0011 PID
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii0111 PORT
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii001011 ATOM
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii011011 CATCH
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii111011 NIL (i always zero...)
iiiiiiiiiiiiiiiiiiiiiiiiiiiiii1111 SMALL_INT
aaaaaaaaaaaaaaaaaaaaaaaaaa000000 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
```

ERTS as memory areas:

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...

C-stack

Queues

Binaries

Processes

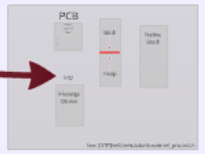
P1

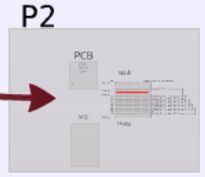
P2

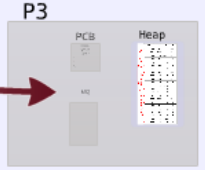
P3

...

P99







...

BEAM CODE

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

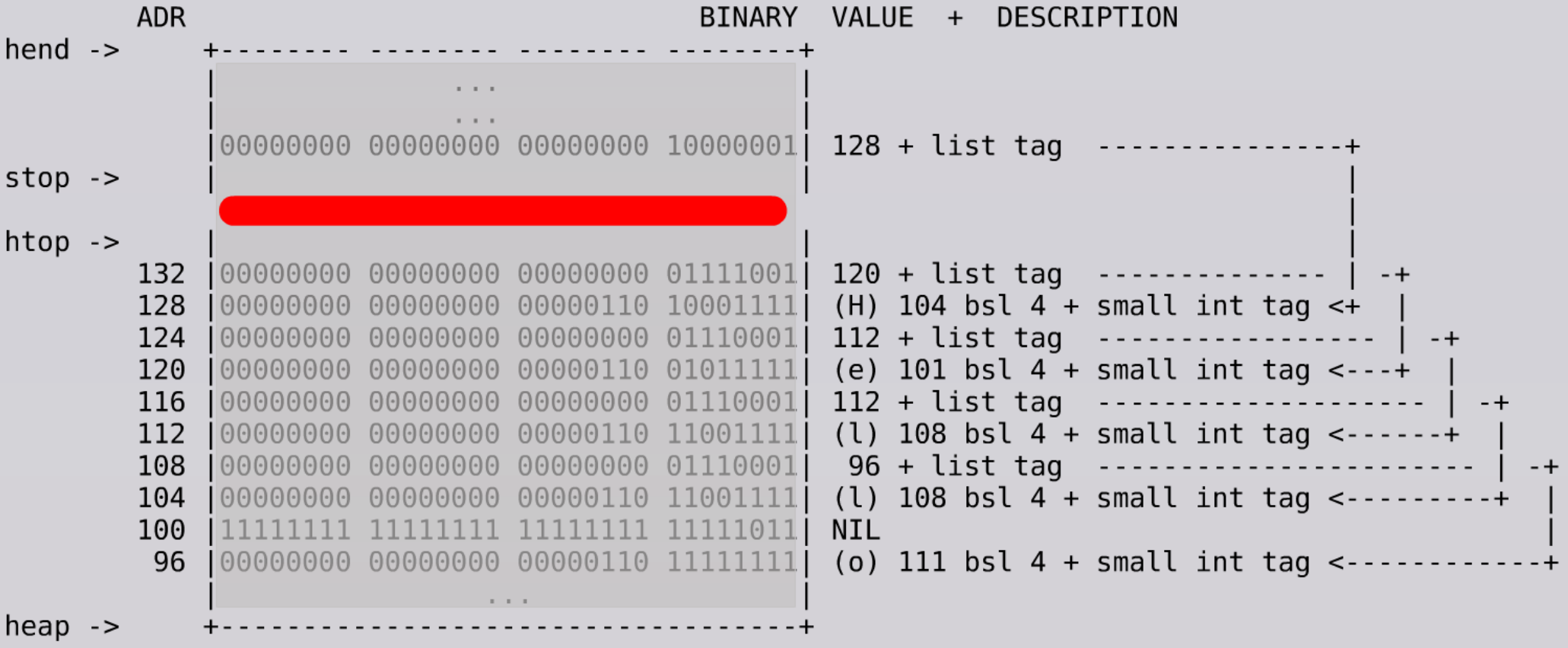
```
p2() ->
    L = "Hello",
    T = {L, L},
    P3 = mk_proc(),
    P3 ! T.
```

An example

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

[104, 101, 108, 108, 111]

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

Sharing

If the list is used in several places all that is repeated is the tagged pointer to the list:

```
0000000000000000000000001000001
```

L = [104, 101, 108, 108, 111],
T = {L, L}.

If the list is used in
several places all that
is repeated is the tagged
pointer to the list:

```
00000000000000000000001000001
```

```
L = [104, 101, 108, 108, 111],
T = {L, L}.
```

ADR	BINARY VALUE	DESCRIPTION
144	1010000 00000000000000000000001000001	128+CONS
140	1001100 00000000000000000000001000001	128+CONS
136	1010100 00000000000000000000010000000	2+ARITYVAL

This is nice...

... until you do

a send
or IO (any deep copy)

... then the sharing
is expanded.

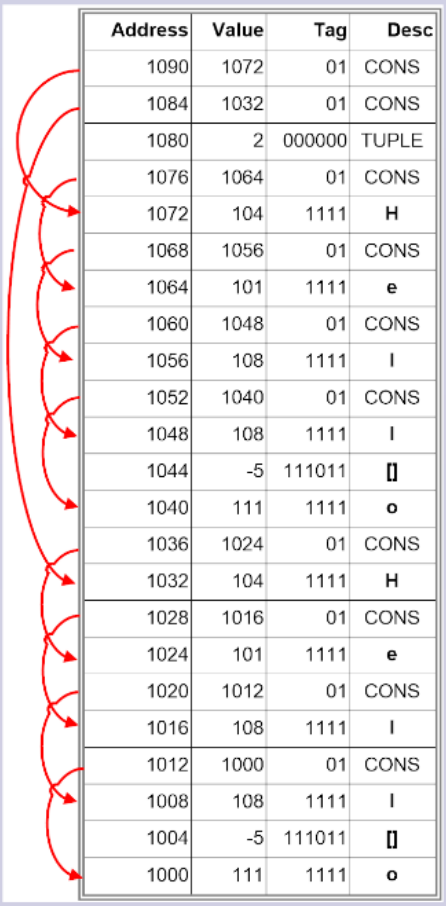
```
p2() ->
    L = "Hello",
    T = {L, L},
    P3 = mk_proc(),
    P3 ! T.
```

PCB

Process Control Block

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

Heap



Address	Value	Tag	Desc
1090	1072	01	CONS
1084	1032	01	CONS
1080	2	000000	TUPLE
1076	1064	01	CONS
1072	104	1111	**H**
1068	1056	01	CONS
1064	101	1111	**e**
1060	1048	01	CONS
1056	108	1111	**l**
1052	1040	01	CONS
1048	108	1111	**l**
1044	-5	111011	**[]**
1040	111	1111	**o**
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**

MQ

Address	Value	Tag	Desc
1090	1072	01	CONS
1084	1032	01	CONS
1080	2	000000	TUPLE
1076	1064	01	CONS
1072	104	1111	**H**
1068	1056	01	CONS
1064	101	1111	**e**
1060	1048	01	CONS
1056	108	1111	**l**
1052	1040	01	CONS
1048	108	1111	**l**
1044	-5	111011	**[]**
1040	111	1111	**o**
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**

```
share(0, Y) -> {Y,Y};
share(N, Y) -> [share(N-1, [N|Y]) || _ <- Y].

 timer:tc(fun() -> test:share(10,[a,b,c]), ok end).
 {1131,ok}

 test:share(10,[a,b,c]), ok.
 ok

 byte_size(list_to_binary(test:share(10,[a,b,c]))), ok.
 HUGE size (13695500364)
 Abort trap: 6
```

ERTS as memory areas:

ERTS CODE

GC
Scheduler
BEAM
Sockets
etc...

C-stack

Queues

Binaries

Processes

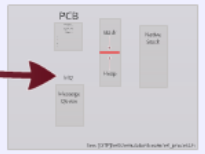
P1

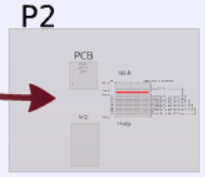
P2

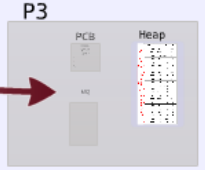
P3

...

P99







...

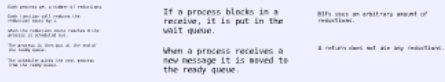
BEAM CODE

```
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 mem areas and a pointer:

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

HiPE

I/O

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

```
c(test, ['S']).
```

The BEAM interpreter

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

Each process get a number of reductions.

Each function call reduces the
reduction count by 1.

When the reduction count reaches 0 the
process is scheduled out.

The process is then put at the end of
the ready queue.

The scheduler picks the next process
from the ready queue.

If a process blocks in a
receive, it is put in the
wait queue.

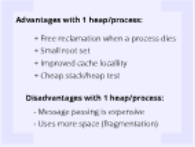
When a process receives a
new message it is moved to
the ready queue.

BIFs uses an arbitrary amount of
reductions.

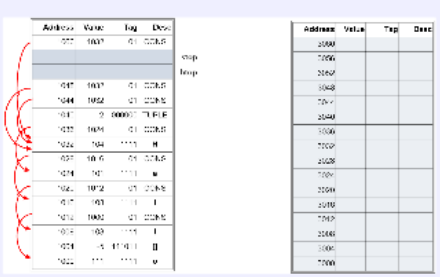
A return does not use any reductions.

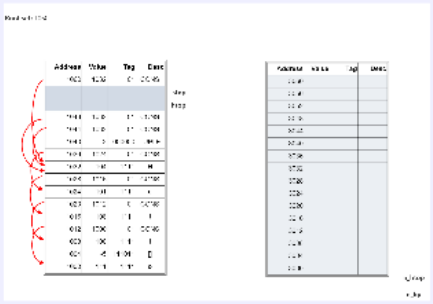
The Garbage Collector

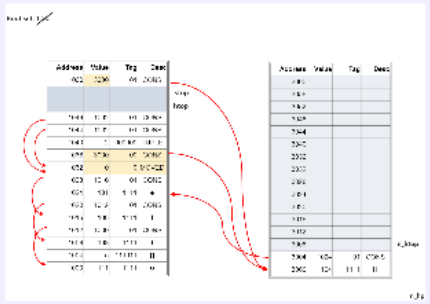
Copying Generational Garbage Collector

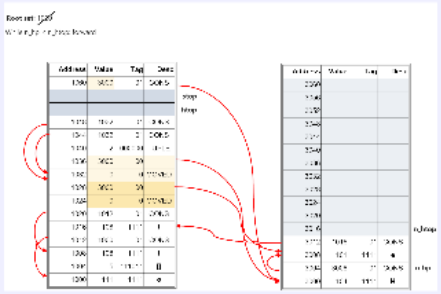


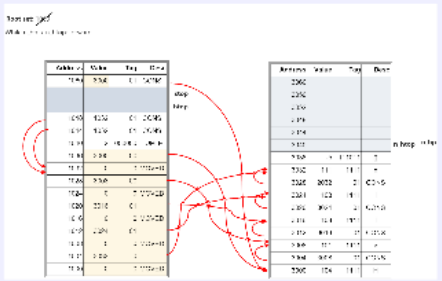
Copying Generational Garbage Collector

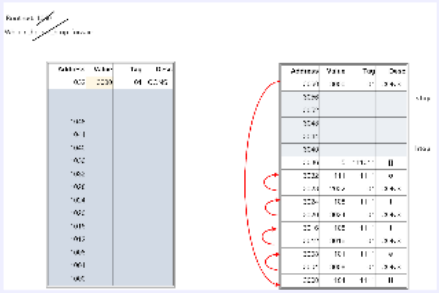












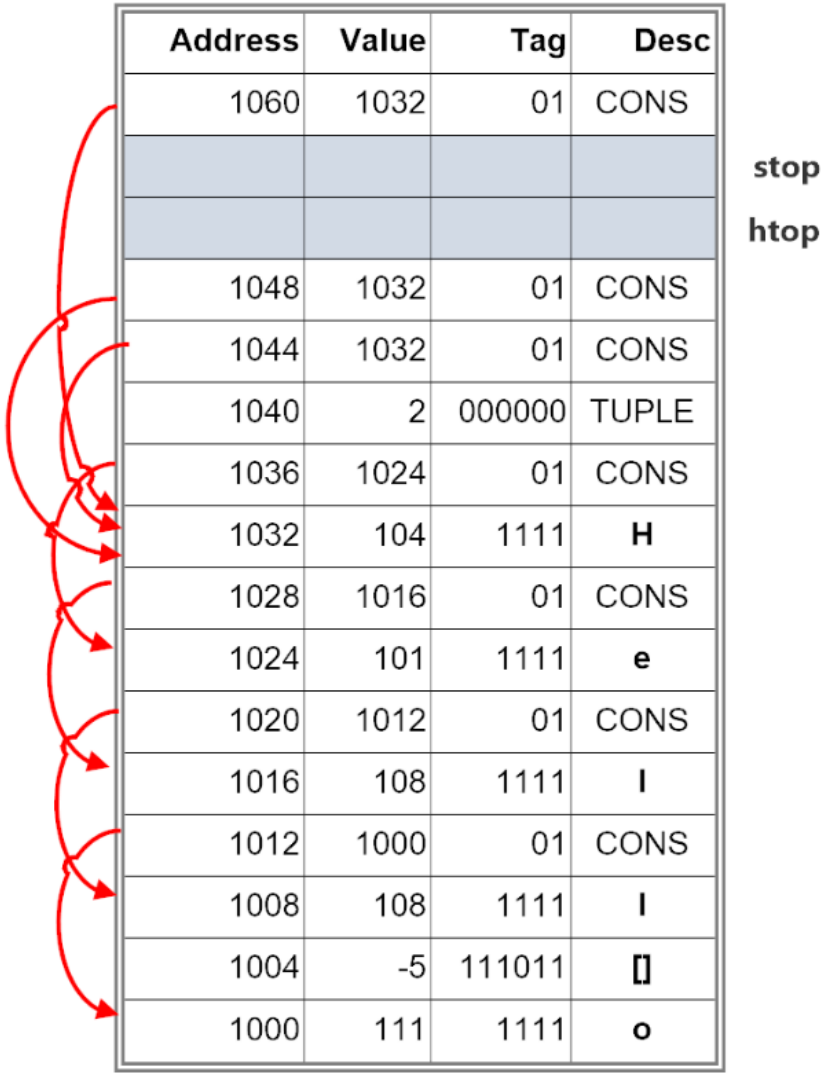
Address	Value	Tag	Desc
1060	1032	01	CONS
1048	1032	01	CONS
1044	1032	01	CONS
1040	2	000000	TUPLE
1036	1024	01	CONS
1032	104	1111	**H**
1028	1016	01	CONS
1024	101	1111	**e**
1020	1012	01	CONS
1016	108	1111	**l**
1012	1000	01	CONS
1008	108	1111	**l**
1004	-5	111011	**[]**
1000	111	1111	**o**

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

stop

htop

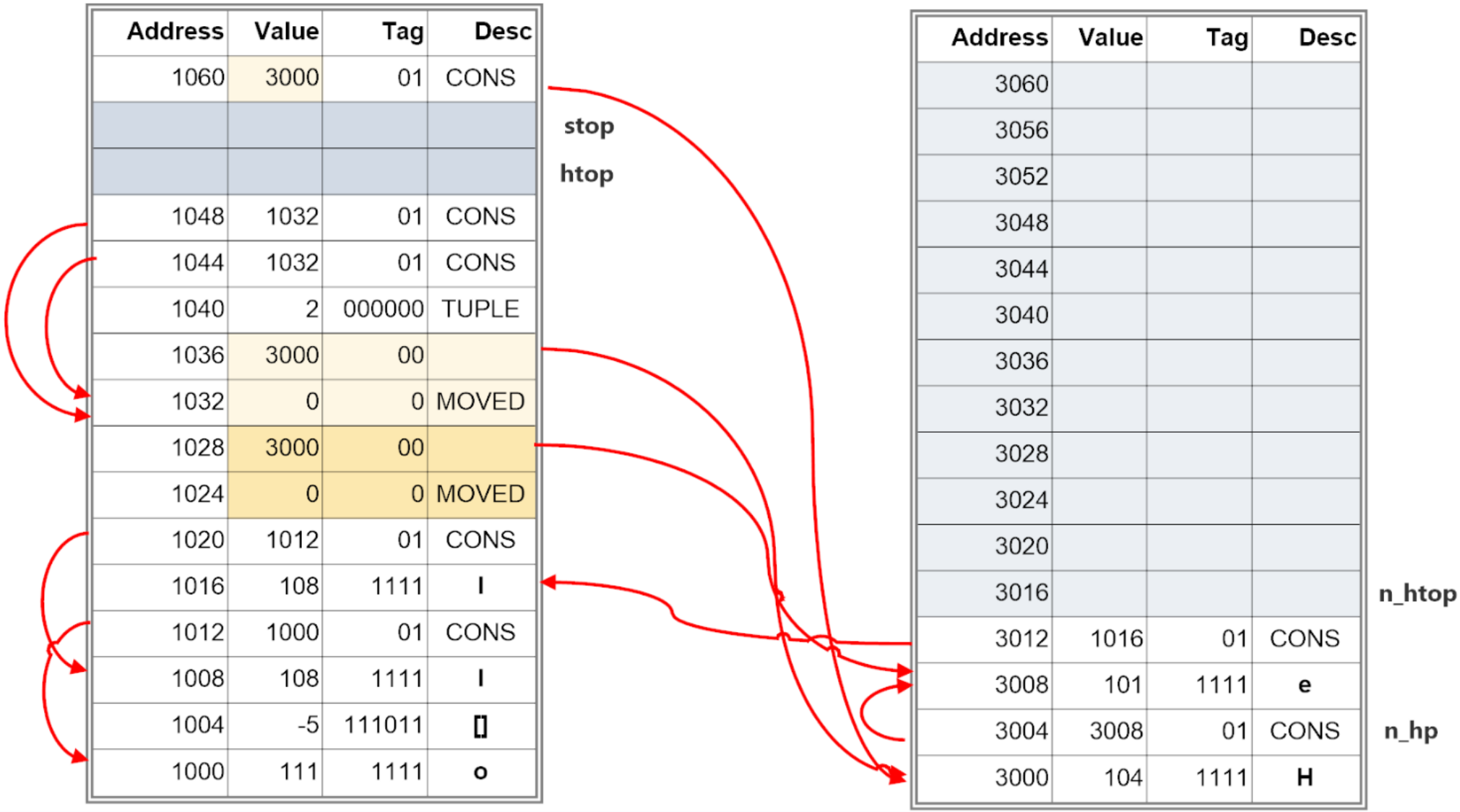
Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036			
3032			
3028			
3024			
3020			
3016			
3012			
3008			
3004	1024	01	CONS
3000	104	1111	H

n_htop

n_hp

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

stop

htop

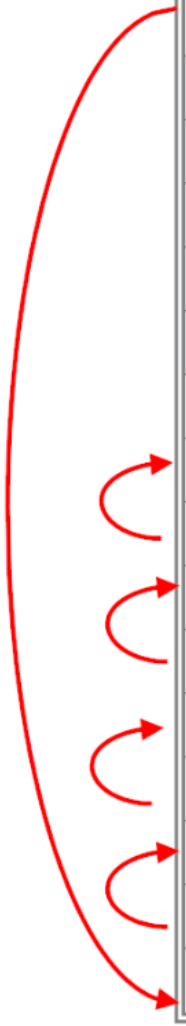
Address	Value	Tag	Desc
3060			
3056			
3052			
3048			
3044			
3040			
3036	-5	111011	[]
3032	111	1111	o
3028	2032	01	CONS
3024	108	1111	l
3020	3024	01	CONS
3016	108	1111	l
3012	3016	01	CONS
3008	101	1111	e
3004	3008	01	CONS
3000	104	1111	H

n_htop n_hp

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	

Erlang has no updates -
there can be no cycles: use reference count.

Why copying collector?

Erlang terms are small.

The HiPE group did some measures:

75% cons cells

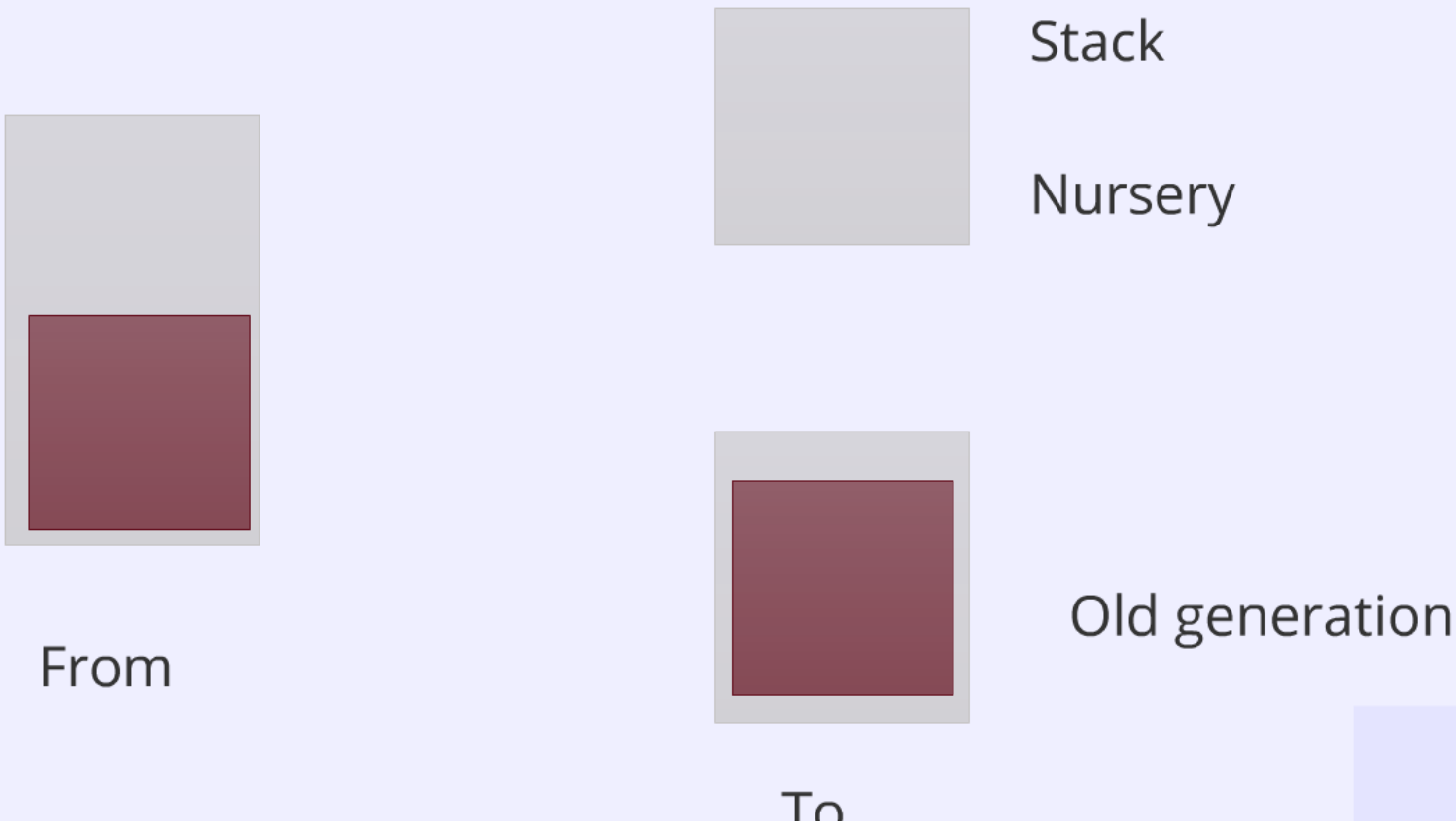
24% !cons but smaller than 8 words

1% >= 8 words

Less fragmentation & better locality with copying collector

Generational GC

"Most objects die young."



Advantages with 1 heap/process:

+ Free reclamation when a process dies
+ Small root set
+ Improved cache locallity
+ Cheap stack/heap test

Disadvantages with 1 heap/process:

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

Binaries are reference counted

`If they are larger than 64 bytes`

Each process has a list of off-heap binaries.
After a GC the reference count is adjusted.

`Two subtle problems:`

If you create a sub-binary the process still has a reference to the binary.

Passing a binary to a process that sends it on without looking at it, creates a new reference.

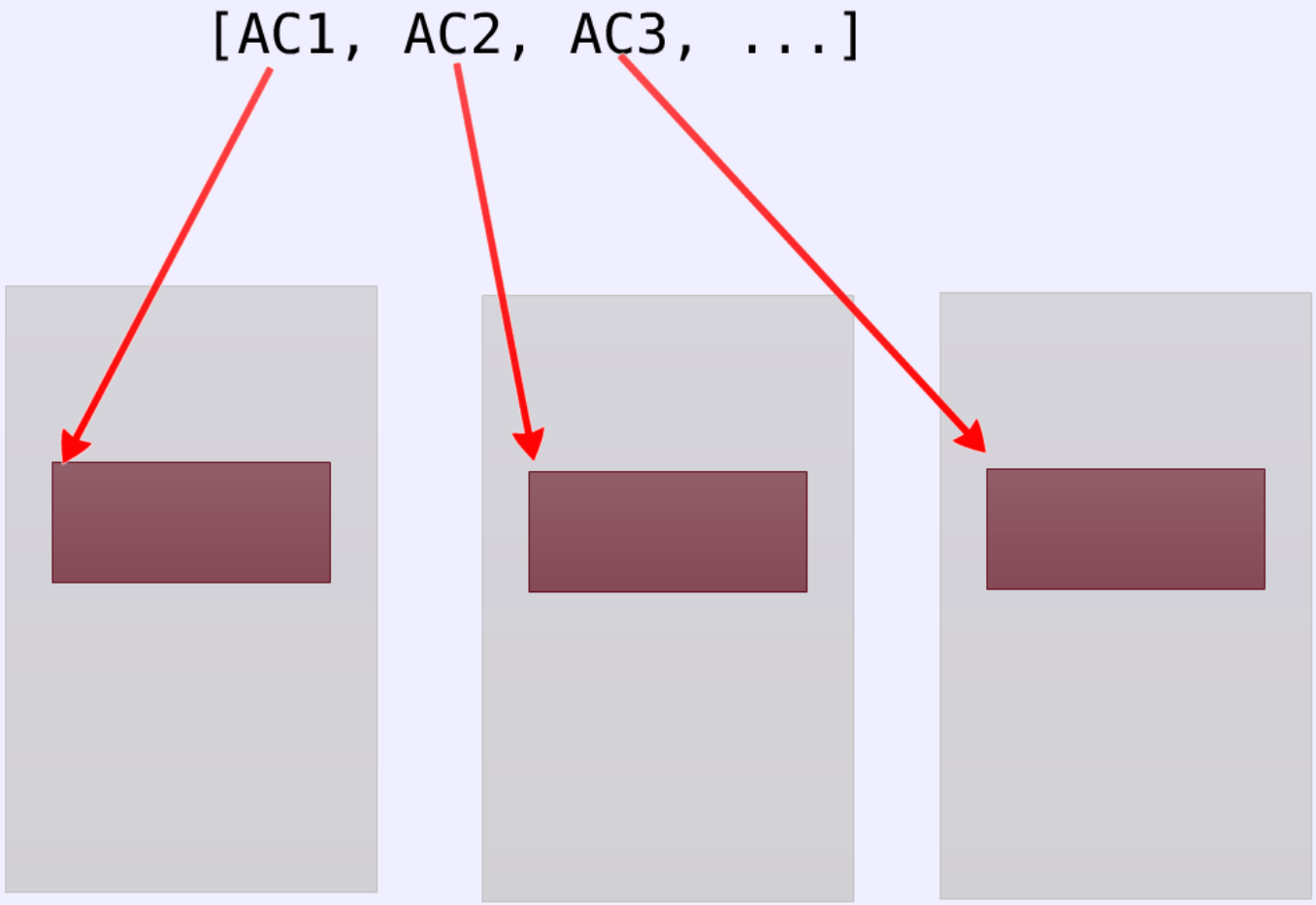
```
-module(beamfile).
-export([read/1]).
read(Filename) ->
  {ok, File} = file:read_file(Filename),
   <<"FOR1", Size:32/integer, "BEAM", Chunks/binary>> = File,
   {Size, read_chunks(Chunks, [])}.

read_chunks(<<N,A,M,E, Size:32/integer, Tail/binary>>, Acc) ->
   %% Align each chunk on even 4 bytes
   ChunkLength = align_by_four(Size),
   <<Chunk:ChunkLength/binary, Rest/binary>> = Tail,
   read_chunks(Rest, [{[N,A,M,E], Size, Chunk}|Acc]);
read_chunks(<<>>, Acc) ->
  lists:reverse(Acc).

align_by_four(N) -> (4 * ((N+3) div 4)).
```

```
get_all_atom_chunks() ->
  [get_atom_chunk(Name) || Name <- all_beam_files()].

get_atom_chunk(FileName) ->
  {_, Chunks} = beamfile:read(FileName),
  {value, {_,_,AtomChunk}} = lists:keysearch("Atom",1,Chunks),
  AtomChunk.
```

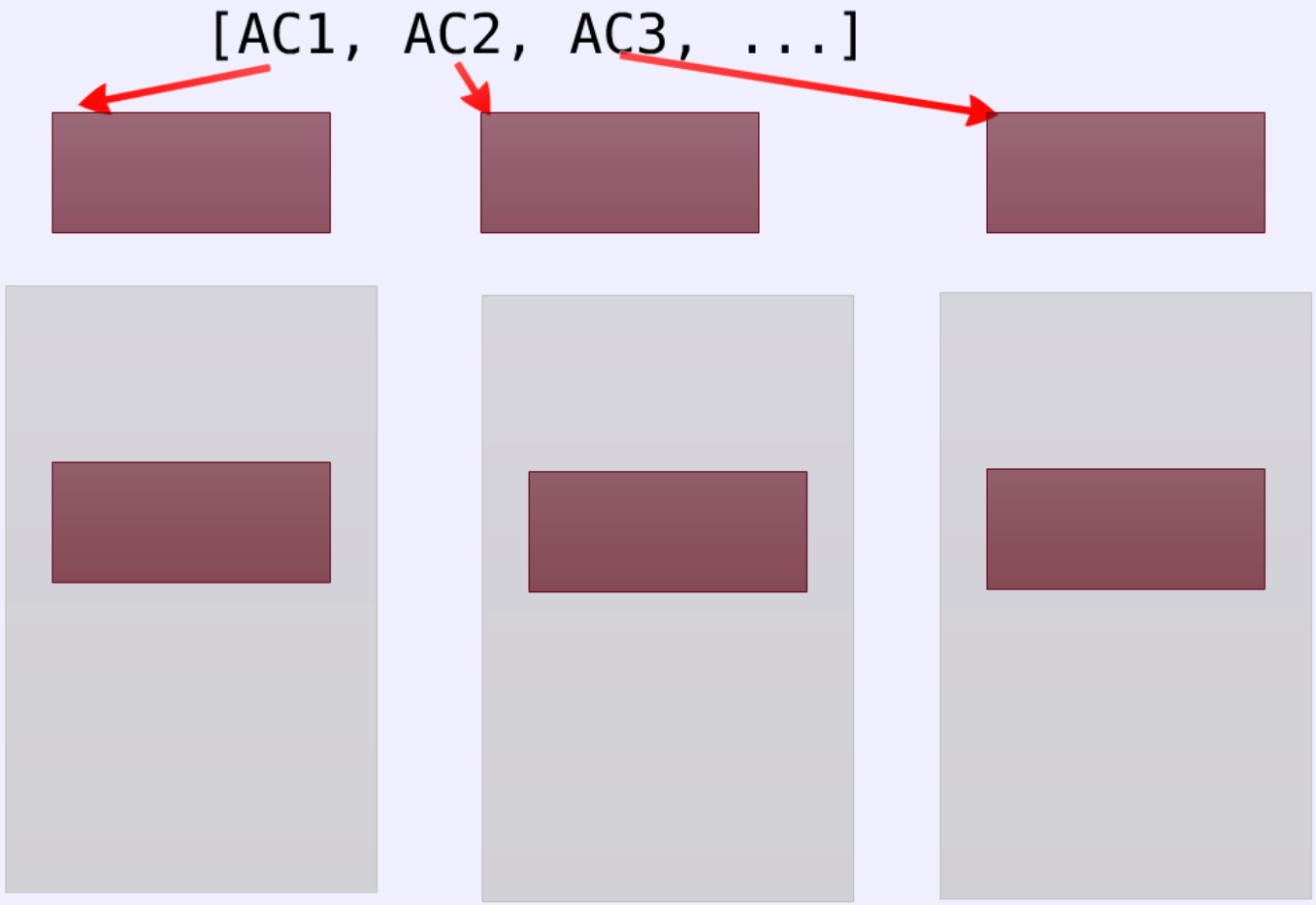


As long as we hang on to the sub-binaries, the GC can't reclaim the parent binaries.

Fix this with binary:copy/1

```
...
read_chunks(<<N,A,M,E, Size:32/integer, Tail/binary>>, Acc) ->
  %% Align each chunk on even 4 bytes
  ChunkLength = align_by_four(Size),
  <<Chunk:ChunkLength/binary, Rest/binary>> = Tail,
  read_chunks(Rest, [{[N,A,M,E], Size, Chunk}|Acc]);
...
```

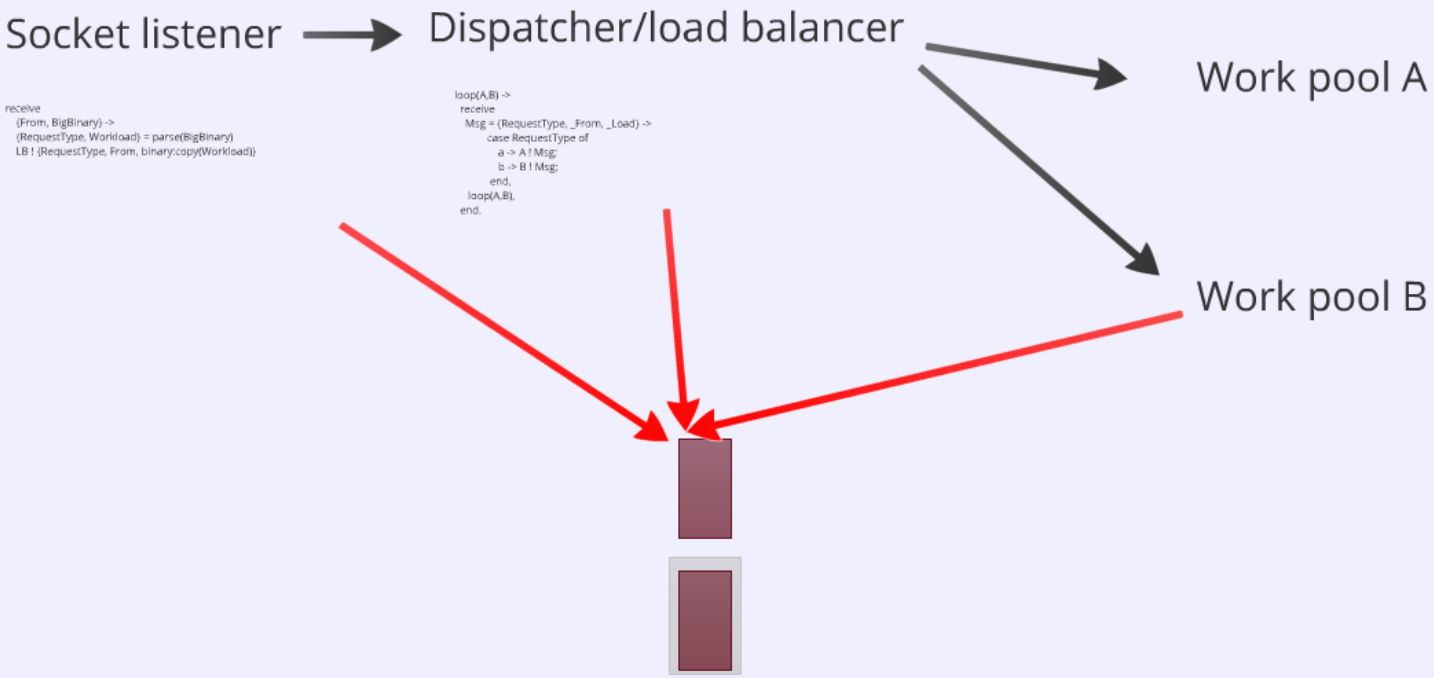
```
...
read_chunks(<<N,A,M,E, Size:32/integer, Tail/binary>>, Acc) ->
  %% Align each chunk on even 4 bytes
  ChunkLength = align_by_four(Size),
  <<Chunk:ChunkLength/binary, Rest/binary>> = Tail,
  read_chunks(Rest, [{[N,A,M,E], Size, binary:copy(Chunk)}|Acc]);
...
```



Now there are no references to the parent binaries and they can be reclaimed.

Problem two

Your nice server architecture:



Socket listener

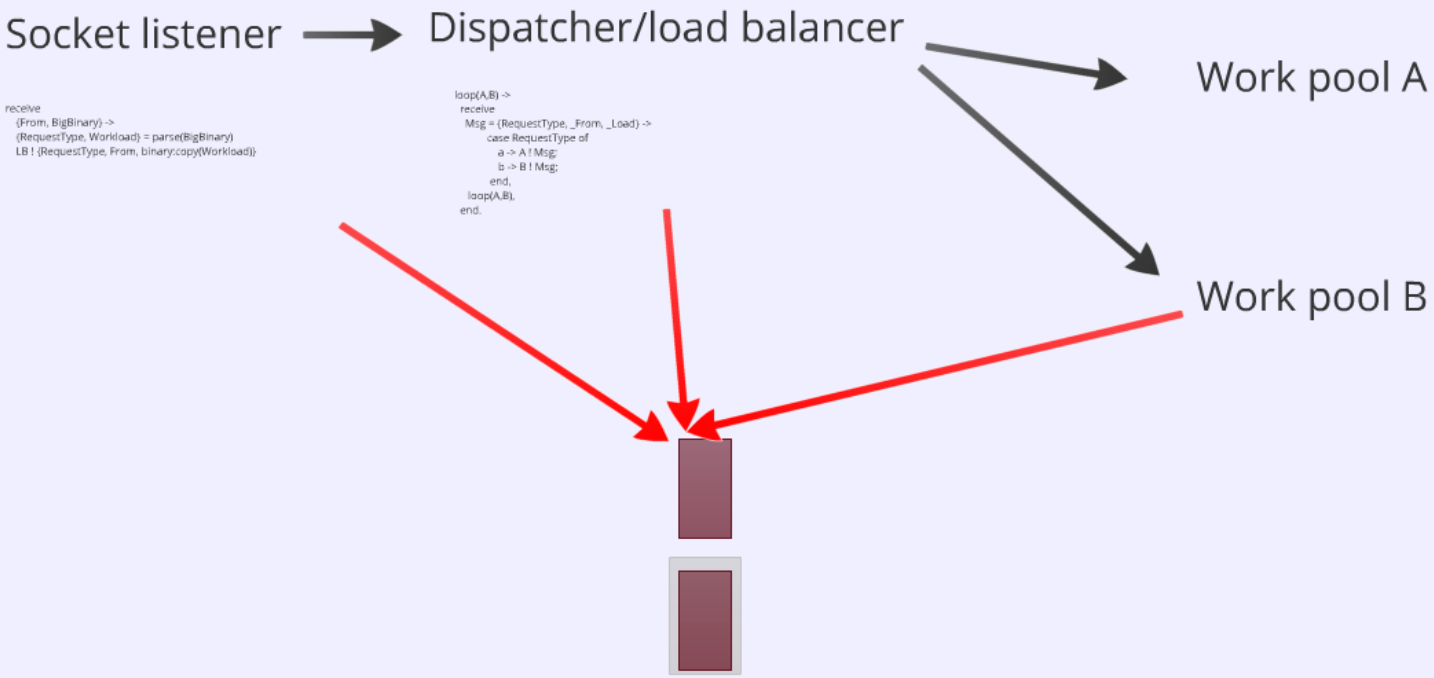
```
receive
    {From, BigBinary} ->
    {RequestType, Workload} = parse(BigBinary)
    LB ! {RequestType, From, binary:copy(Workload)}
```

Dispatcher/load balancer

```
loop(A,B) ->
  receive
    Msg = {RequestType, _From, _Load} ->
          case RequestType of
             a -> A ! Msg;
             b -> B ! Msg;
          end,
     loop(A,B),
  end.
```

Problem two

Your nice server architecture:



Dispatcher/load balancer

```
loop(A,B) ->
  receive
    Msg = {RequestType, _From, _Load} ->
          case RequestType of
             a -> A ! Msg;
             b -> B ! Msg;
          end,
     loop(A,B),
  end.
```

Solution

Add an explicit GC

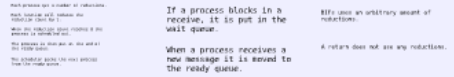
Dispatcher/load balancer

```
loop(A,B) ->
  receive
    Msg = {RequestType, _From, _Load} ->
          case RequestType of
             a -> A ! Msg;
             b -> B ! Msg;
          end,
     loop(A,B);
  after 1000 ->
    garbage_collect(),
    loop(A,B)
  end.
```

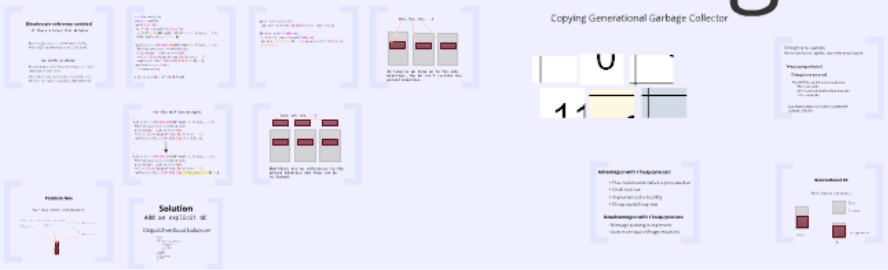
ERTS as components:

The BEAM interpreter

The Scheduler



The Garbage Collector



Processes

Conceptually: 4 mem areas and a pointer:

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

HiPE

I/O

Lessons learned:

- Processes starts with small heaps
- Strings takes space
- Sharing is nice but can explode
- A copying GC uses twice the space
- Binaries are reference counted
- Every process who has seen a binary has to forget it, before it goes away.
- Use binary:copy/1 on sub binaries


QUESTIONS?