

Erlang Engine Tuning: Know Your Engine Part 2: The Beam



What is ERTS?

ERTS is the Erlang Runtime System.

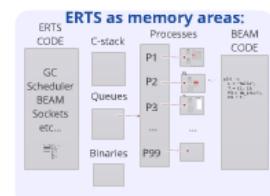


ERTS as source code:

See: [OTP/erts/
emulator/
beam/
hipec/
etc/

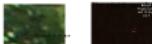
ERTS as components:

The BEAM interpreter
The Scheduler
The Garbage Collector
HIPE
I/O



QUESTIONS?

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



Lessons learned:

- BEAM is a...
- Garbage Collecting
- Reductive Computing
- Concurrent
- Dose of threads
- Registers
- Virtual Machine
- Use the smp TS library to get better data



Erlang Engine Tuning: Know Your Engine Part 2: The Beam



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

A portrait photo of Erik Stenman, a man with a beard and short hair, smiling. He is wearing a dark jacket over a light-colored shirt. The background is a plain yellow wall.

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/

ERTS as components:

The BEAM interpreter



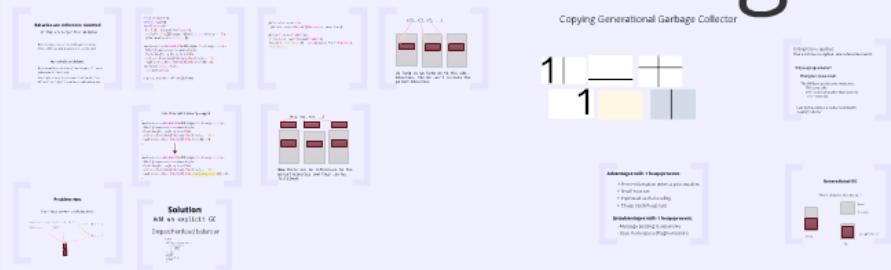
The Scheduler

RIFs are scheduled in a queue of ready processes. If a process is suspended, it is moved to the ready queue. When a process resumes, it is moved to 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.

A return does not use any reductions.

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

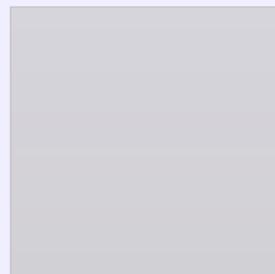
A PID

ERTS as memory areas:

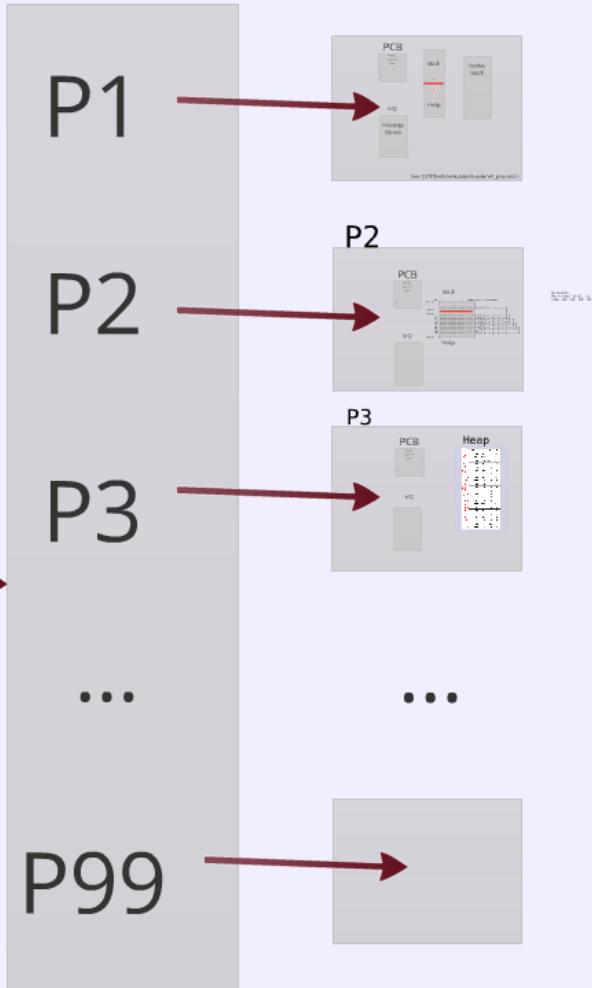
ERTS
CODE



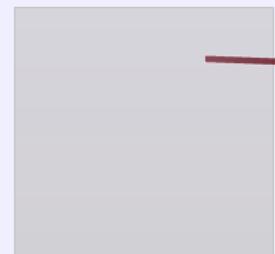
C-stack



Processes



Queues



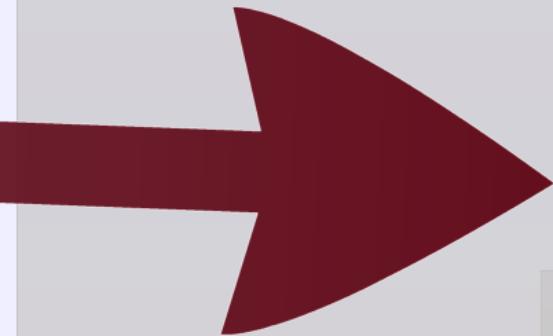
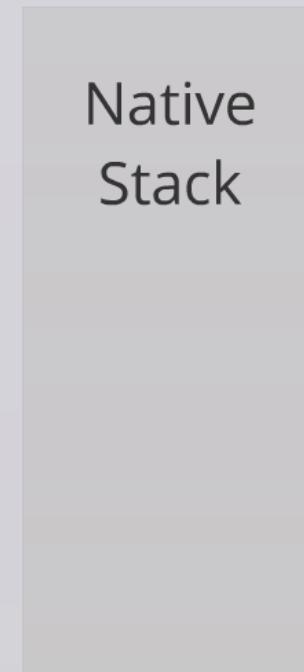
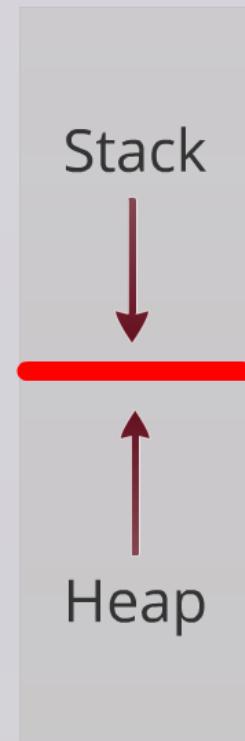
Binaries



BEAM
CODE

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

PCB



MQ

Message
Queue

See: [OTP]/erts/emulator/beam/erl_process.h

ERTS

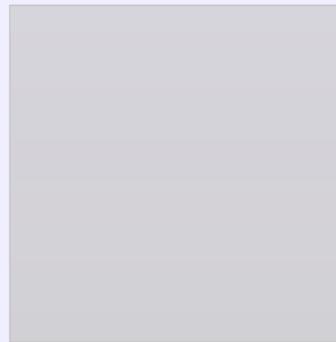
CODE

GC
Scheduler
BEAM
Sockets
etc...

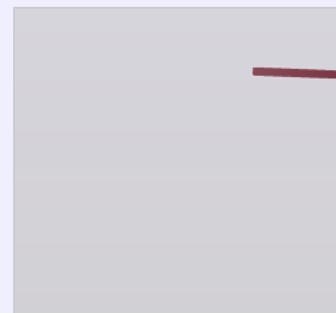
The Tag Scheme



C-stack



Queues



Binaries



Proce

P1

P2

P3

...

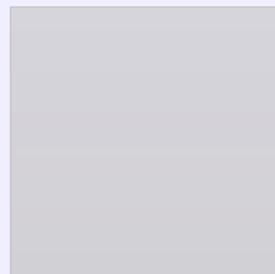
P99

ERTS as memory areas:

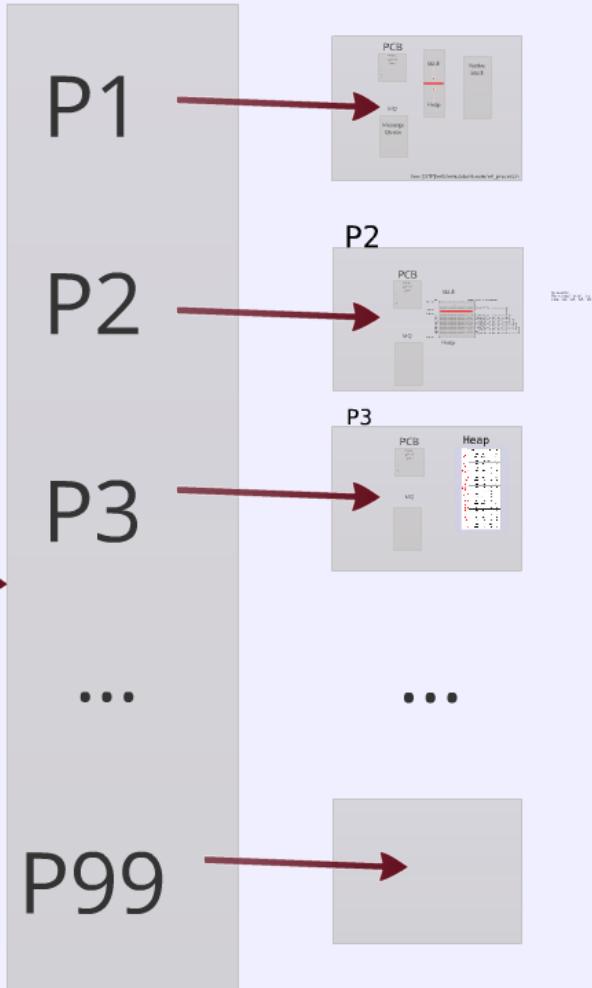
ERTS
CODE



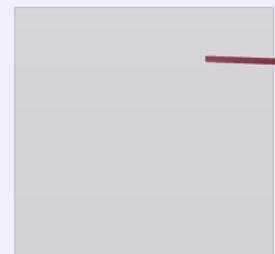
C-stack



Processes



Queues



Binaries



BEAM
CODE

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

ERTS as components:

The BEAM interpreter



BEAM
• Garbage Collector
• Virtual Machine
• Network Driver
• Direct Threads
• Debugger
• VMFS
Allocated

The Scheduler

If a process sends a message to another process, it is added to the ready queue. If a process receives a message, it is moved to the ready queue. If a process receives a new message, it is moved to the ready queue. If a process does not receive any messages, it remains in the ready queue.

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



BEAM is Virtually Unreal

The Beam is a virtual machine: it is implemented in software instead of in hardware.

There is no official specification of the Beam, it is currently only defined by the implementation in Erlang/OTP.

A Stack Machine - it is not

BEAM is a register machine

Advantage of a stack machine

- Easier to compile to
- Easier to implement

See my blog: http://stenmans.org/happi_blog/?p=194
for an example of a stack machine.

Advantage of a register machine

- More efficient (?)

- Two types of registers: X and Y-registers.
- X0 is the accumulator and mapped to a physical register, also called R0.
- Y registers are actually stack slots.

There are a number of special purpose registers:
htop, stop, l, fcalls and floating point registers.

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);  
}
```

External beam format:

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

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

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

Loaded code*:

0x1000: 0x3000



0x3000:

0x1004: 0x0

0x1008: 0x1

0x100c: 0x3200



0x3200:

0x1010: 0x1

0x1014: 0x1

0x1018: 0x3100



Op

0x101c: 0x1

0x3100: y

0x1020: 0x1

I

at:

1}}).

0}}).

0}}).

actually rewrites the
different internal

beam_emu.c **:

OpCase(move_xx): {

0x3000: x(Arg(1)) = x(Arg(0));

 | += 3;

 Goto(*|);

}

** This is another lie, there are no such
instructions in beam_emu, but you can't handle
the truth.

OpCase(move_yx): {

0x3200: y(Arg(1)) = y(Arg(0));

e.

```
#define Arg(N) (Eterm *) I[(N)+1]
```

```
#define Goto(Rel) goto *((void *)Rel)
```

beam_emu.c **:

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

** This is another lie, there are no such instructions in beam_emu, but you can't handle the truth.

```
    OpCase(move_vx): {
```

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

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));
I += 3;
Goto(*I);
}
```

** This is another lie, there are no such instructions in beam_emu, but you can't handle the truth.

```
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);
}
```

Scheduling:

Non-preemptive, Reduction counting

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

A process can also yield in a receive.

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.

BEAM

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

-Machine



world.hrl

```
-define(GREETING, "hello world").
```

world.erl

```
-module(world).
-export([hello/0]).
```

```
-include("world.hrl").
```

```
hello() -> ?GREETING.
```

```
1> c(world, ['P']).  
** Warning: No object file created - nothing loaded **  
ok
```

world.P

```
-file("world.erl",1).
-module(world).
-export([hello/0]).
-file("world.hrl", 1).
-file("world.erl", 4).
hello() ->
    "hello world".
```

1> c(world, 'S').

```
{module, world}.
%% version = 0
{exports, [{hello,0}, {module_info,0},
{module_info,1}]}.
{attributes, []}.
{labels, 7}.
```

```
{function, hello, 0, 2}.
{label,1}.
{func_info, {atom,world}, {atom,hello}, 0}.
{label,2}.
{move,{literal,"hello world"},{x,0}}.
return.
```

```
{function, module_info, 0, 4}.
{label,3}.
{func_info, {atom,world}, {atom,module_info},0}.
{label,4}.
{move,{atom,world},{x,0}}.
{call_ext_only, 1,
    {extfunc, erlang, get_module_info, 1}}.
```

```
{function, module_info, 1, 6}
{label,5}.
{func_info,{atom,world},{atom,module_info},1}.
{label,6}.
{move,{x,0},{x,1}}.
{move,{atom,world},{x,0}}.
{call_ext_only, 2,
    {extfunc, erlang, get_module_info, 2}}.
```

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

world.hrl

```
-define(GREETING, "hello world").
```

world.erl

```
-module(world).  
-export([hello/0]).  
  
-include("world.hrl").  
  
hello() -> ?GREETING.
```

hello() -> ?GREETING.

1> c(world, ['P']).

** Warning: No object file created - nothing loaded **
ok

world.P

file("world.orl" 1)

```
-file("world.erl",1).
-module(world).
-export([hello/0]).
-file("world.hrl", 1).
-file("world.erl", 4).
hello() ->
    "hello world".
```

1> c(world, 'S').

{module, world}.

%% version = 0

```
{module, world}.
```

```
%% version = 0
```

```
{exports, [{hello,0}, {module_info,0},
```

```
{module_info,1}]}.
```

```
{attributes, []}.
```

```
{labels, 7}.
```



```
{function, hello, 0, 2}.
```

```
{label,1}.
```

```
{func_info, {atom,world}, {atom,hello}, 0}.
```

```
{label,2}.
```

```
{move,{literal,"hello world"},{x,0}}.
```

```
return.
```



```
{function, module_info, 0, 4}.
```

```
{label,3}.
```

```
{func_info, {atom,world}, {atom,module_info},0}.
```

```
{label,4}.
```

```
{move,{atom,world},{x,0}}.
```

```
{call_ext_only, 1,
```

```
    {extfunc, erlang, get_module_info, 1}}.
```



```
{function, module_info, 1, 6}
```

```
{label,5}.
```

```
{func_info,{atom,world},{atom,module_info},1}.
```

```
{label,6}.
```

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

```
{move,{atom,world},{x,0}}.
```

```
{call_ext_only, 2,
```

```
    {extfunc, erlang, get_module_info, 2}}.
```

```
{label,5}.
```

```
{func_info,{atom,world},{atom,module_info},1}.
```

```
{label,6}.
```

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

```
{move,{atom,world},{x,0}}.
```

```
{call_ext_only, 2,
```

```
    {extfunc, erlang, get_module_info, 2}}.
```

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

Beam Instructions

An Added Example

```
-module(add).  
-export([add/2]).
```

```
add(A,B) -> id(A) + id(B).
```

```
id(I) -> I.
```

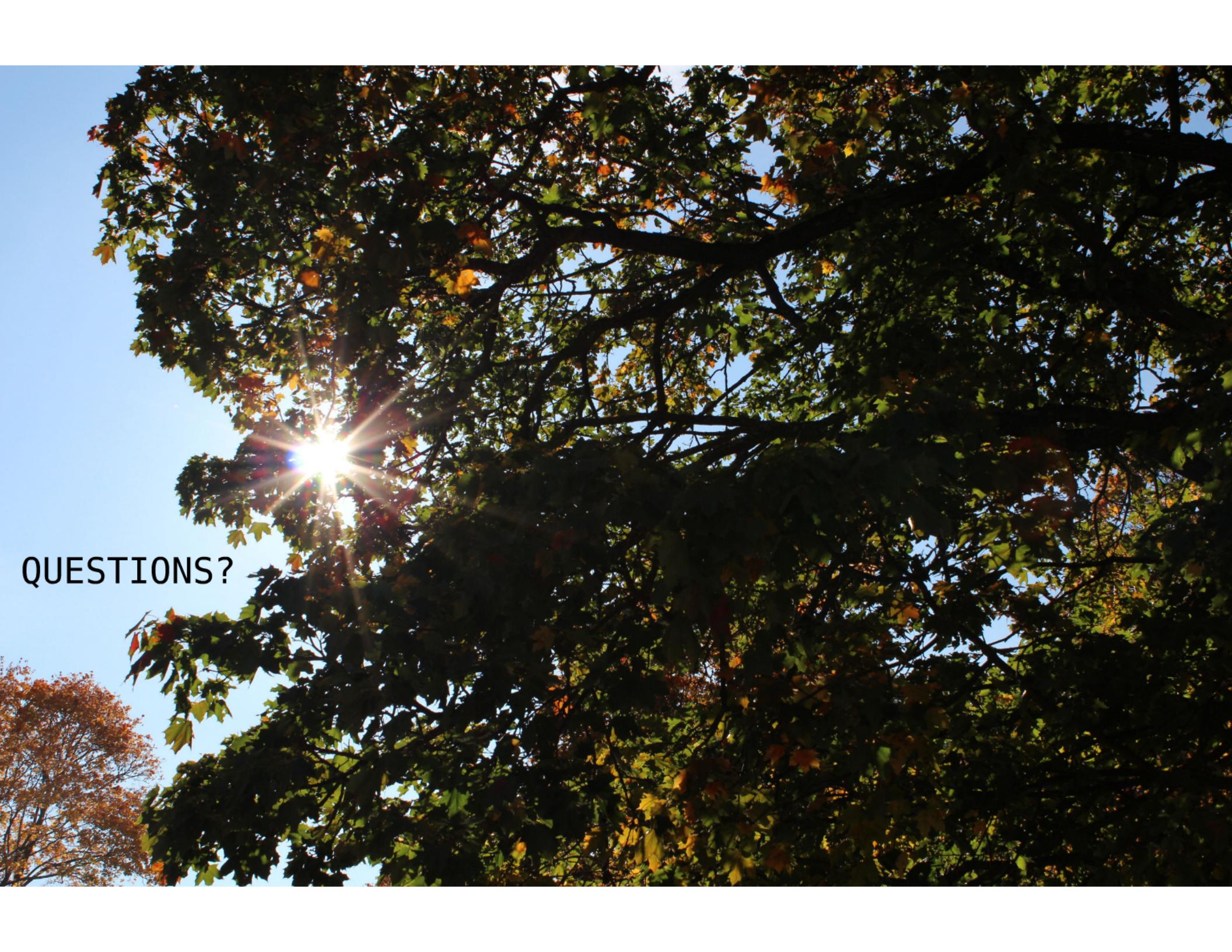
```
{allocate,1,2}.  
{move,{x,1},{y,0}}.  
{call,1,{f,4}}.  
{move,{x,0},{x,1}}.  
{move,{y,0},{x,0}}.  
{move,{x,1},{y,0}}.  
{call,1,{f,4}}.  
{gc_bif,'+',{f,0},1,[{y,0},{x,0}],{x,0}}.  
{deallocate,1}.  
return.
```



Lessons learned:

- BEAM is a
 - Garbage Collecting-
 - Reduction Counting-
 - Non-preemptive-
 - Directly Threaded-
 - Register-
 - Virtual- -Machine
- Use the option ['S', binary] to get beam code.



A photograph of a large tree with dense foliage. The leaves are primarily green, with some yellow and orange ones interspersed, suggesting autumn. The tree is set against a clear blue sky. Sunlight filters through the branches, creating bright, radial lens flares that radiate outwards from the left side of the frame.

QUESTIONS?