The Erlang Runtime System

COLLABORATORS			
	TITLE : The Erlang Runtime System		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 8, 2017	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

I Understanding ERTS 1

1 Introducing the Erlang Runtime System 2

1.1 ERTS and the Erlang Runtime System . . . 2
1.2 How to read this book . . . 2
1.3 ERTS . . . 3
1.3.1 The Erlang Node (ERTS) . . . 3
1.3.2 The Erlang Compiler . . . 4
1.3.3 The Erlang Virtual Machine: BEAM . . . 4
1.3.4 Processes . . . 4
1.3.5 Scheduling . . . 4
1.3.6 The Erlang Tag Scheme . . . 5
1.3.7 Memory Handling . . . 5
1.3.8 The Interpreter and the Command Line Interface . . . 5
1.4 Other Erlang Implementations . . . 5
1.4.1 Erlang on Xen . . . 6
1.4.2 Erjang . . . 6

2 The Compiler 7

2.1 Compiling Erlang . . . 7
2.2 Generating Intermediate Output . . . 9
2.3 Compiler Passes . . . 12
2.3.1 Compiler Pass: The Erlang Preprocessor (epp) . . . 12
2.3.2 Compiler Pass: Parse Transformations . . . 12
2.3.3 Compiler Pass: Linter . . . 16
2.3.4 Compiler Pass: Save AST . . . 16
2.3.5 Compiler Pass: Expand . . . 16
2.3.6 Compiler Pass: Core Erlang . . . 16
2.3.7 Compiler Pass: Kernel Erlang . . . 17
2.3.8 Compiler Pass: BEAM Code . . . 17

2.3.9 Compiler Pass: Native Code . . . 17
2.4 Other Compiler Tools . . . 17
2.4.1 Leex . . . 17
2.4.2 Yecc . . . 21
2.5 Syntax Tools and Merl . . . 22
2.6 Compiling Elixir . . . 22

3 Processes 23
3.1 What is a Process? . . . 23
3.1.1 Listing Processes from the Shell . . . 23
3.1.2 Programatic Process Probing . . . 25
3.1.3 Using the Observer to Inspect Processes . . . 27
3.2 Processes Are Just Memory . . . 27
3.3 The PCB . . . 30
3.4 The Garbage Collector (GC) . . . 31
3.5 Mailboxes and Message Passing . . . 32
3.5.1 Sending Messages in Parallel . . . 32
3.6 Lock Free Message Passing . . . 32
3.6.1 Memory Areas for Messages . . . 33
3.6.2 Inspecting Message Handling . . . 33
3.6.3 The Process of Sending a Message to a Process . . . 34
3.6.4 Receiving a Message . . . 37
3.6.5 Tuning Message Passing . . . 37
3.7 The Process Dictionary . . . 38
3.8 Dig In . . . 38

4 The Erlang Type System and Tags 40
4.1 The Erlang Type System . . . 40
4.2 The Tagging Scheme . . . 41
4.2.1 Tags for Immediates . . . 41
4.2.2 Tags for Boxed Terms . . . 42

5 The Erlang Virtual Machine: BEAM 45
5.1 Working Memory: A stack machine, it is not . . . 45
5.2 Dispatch: Directly Threaded Code . . . 47
5.3 Scheduling: Non-preemptive, Reduction counting . . . 52
5.4 Memory Management: Garbage Collecting . . . 52
5.5 BEAM: it is virtually unreal . . . 53

6 Modules and The BEAM File Format (16p) 54

6.1 Modules . . . 54
6.2 The BEAM File Format . . . 54
6.2.1 Atom table chunk . . . 55
6.2.2 Export table chunk . . . 56
6.2.3 Import table chunk . . . 57
6.2.4 Code Chunk . . . 57
6.2.5 String table chunk . . . 58
6.2.6 Attributes Chunk . . . 58
6.2.7 Compilation Information Chunk . . . 59
6.2.8 Local Function Table Chunk . . . 59
6.2.9 Literal Table Chunk . . . 59
6.2.10 Abstract Code Chunk . . . 60
6.2.11 Compression and Encryption . . . 60
6.2.12 Function Trace Chunk (Obsolete) . . . 60
6.2.13 Bringing it all Together . . . 60

7 Generic BEAM Instructions (25p) 61

7.1 Instruction definitions . . . 61
7.2 BEAM code listings . . . 62
7.3 Calls . . . 63
7.4 Stack (and Heap) Management . . . 63
7.5 Message Passing . . . 63
7.5.1 A Minimal Receive Loop . . . 64
7.5.2 A Selective Receive Loop . . . 64
7.5.3 A Receive Loop With a Timeout . . . 65
7.5.4 The Synchronous Call Trick (aka The Ref Trick) . . . 65

8 Different Types of Calls, Linking and Hot Code Loading (5p) 67

8.1 Hot Code Loading . . . 67
8.2 Code Loading . . . 68
8.3 Transforming from Generic to Specific instructions . . . 68

9 The BEAM Loader 69

9.1 Transforming from Generic to Specific instructions . . . 69

10 BEAM Internal Instructions 70

11 Scheduling 71
11.1 Concurrency, Parallelism, and Preemptive Multitasking . . . 71
11.2 Preemptive Multitasking in ERTS Cooperating in C . . . 73
11.3 Reductions . . . 74
11.3.1 How Many Reductions Will You Get? . . . 74
11.3.2 What is a Reduction Really? . . . 74
11.4 The Process State (or *status*) . . . 75
11.5 Process Queues . . . 76
11.5.1 The Ready Queue . . . 76
11.5.2 Waiting, Timeouts and the Timing Wheel . . . 77
11.6 Ports . . . 77
11.7 Reductions . . . 78
11.8 The Scheduler Loop . . . 78
11.9 Load Balancing . . . 79
11.9.1 Task Stealing . . . 79
11.9.2 Migration . . . 79

12 The Memory Subsystem: Stacks, Heaps and Garbage Collection 81
12.1 The memory subsystem . . . 82
12.2 Different type of memory allocators . . . 83
12.2.1 The basic allocator: sys_alloc . . . 84
12.2.2 The memory segment allocator: mseg_alloc . . . 84
12.2.3 The memory allocator framework: alloc_util . . . 84
12.2.3.1 Memory allocation strategies . . . 85
12.2.4 The temporary allocator: temp_alloc . . . 85
12.2.5 The heap allocator: eheap_alloc . . . 86
12.2.6 The binary allocator: binary_alloc . . . 86
12.2.7 The ETS allocator: ets_alloc . . . 86
12.2.8 The driver allocator: driver_alloc . . . 86
12.2.9 The short lived allocator: sl_alloc . . . 86
12.2.10 The long lived allocator: ll_alloc . . . 86
12.2.11 The fixed size allocator: fix_alloc . . . 87
12.2.12 The standard allocator: std_alloc . . . 87
12.3 TODO: system flags for memory . . . 87
12.4 Process Memory . . . 87
12.4.1 Term sharing . . . 87
12.4.2 Message passing . . . 89
12.4.3 Binaries . . . 90
12.4.3.1 Reference Counting . . . 91
12.4.3.2 Sub Binaries and Matching . . . 91
12.4.4 Garbage Collection . . . 92
12.5 Other interesting memory areas . . . 98
12.5.1 The atom table. . . . 98

13 Advanced data structures (ETS, DETS, Mnesia) 99

14 IO, Ports and Networking (10p) 100

14.1 Standard IO . . . 100
14.2 Ports . . . 100
14.2.1 Different types of Ports . . . 101
14.2.1.1 Ports to file descriptors . . . 102
14.2.1.2 Ports to Spawned OS Processes . . . 102
14.2.1.3 Ports to Linked in Drivers . . . 102
14.3 Distributed Erlang . . . 102
14.4 Sockets, UDP and TCP . . . 102

15 Distribution 103

16 Interfacing C — BIFs NIFs and Linked in Drivers 104

17 Native code 105

II Running ERTS 106

18 Operation and Maintenance 107

III Apendicies 108

.1 Building Erlang OTP . . . 109
.1 BEAM Instructions . . . 109
.2 Functions and Labels . . . 109
.2.1 label Lbl . . . 109
.2.2 func_info Module Function Arity . . . 109
.3 Test instructions . . . 109
.3.1 Type tests . . . 109
.3.2 Comparisons . . . 110
.4 Function Calls . . . 110
.4.1 call Arity Label . . . 110
.4.2 call_last Arity Label . . . 110
.4.3 call_only Arity Label Deallocate . . . 110
.4.4 call_ext Arity Destination . . . 110
.4.5 call_ext_only Arity Destination . . . 110
.4.6 call_ext_last Arity Destination Deallocate . . . 111
.4.7 `bif0 Bif Reg`, `bif1 Lbl Bif Arg Reg`, `bif2 Lbl Bif Arg1 Arg2 Reg` . . . 111
.4.8 `gc_bif1-3 Lbl Live Bif Arg1-3 Reg` . . . 111

.4.9 call_fun Arity . . . 111
.4.10 apply/1 . . . 111
.4.11 apply_last/2 . . . 111
.5 Stack (and Heap) Management . . . 111
.5.1 allocate StackNeed Live . . . 112
.5.2 allocate_heap StackNeed HeapNeed Live . . . 112
.5.3 allocate_zero StackNeed Live . . . 112
.5.4 allocate_heap_zero StackNeed HeapNeed Live . . . 113
.5.5 test_heap HeapNeed Live . . . 113
.5.6 init N . . . 113
.5.7 deallocate N . . . 113
.5.8 return . . . 113
.5.9 trim N Remaining . . . 113
.6 Moving, extracting, modifying. . . . 113
.6.1 move Source Destination . . . 113
.6.2 get_list Source Head Tail . . . 113
.6.3 get_tuple_element Source Element Destination . . . 114
.6.4 set_tuple_element NewElement Tuple Position . . . 114
.7 Building terms. . . . 114
.7.1 put_list/3 . . . 114
.7.2 make_fun2/1 . . . 114
.8 Binary Syntax . . . 114
.8.1 bs_put_integer/5 . . . 114
.9 Floating Point Arithmetic . . . 114
.9.1 fclearerror/0 . . . 114
.10 Pattern Matching . . . 114
.10.1 select_val . . . 114
.11 Meta instructions . . . 115
.11.1 on_load . . . 115
.12 Generic Instructions . . . 115
.13 Specific Instructions . . . 123
.14 Full Code Listings . . . 127

1 Index 137

List of Figures

7.1 A simple receive loop. 64
7.2 A selective receive loop. 64
7.3 A receive loop with a timeout. 65
7.4 The Ref Trick Pattern. 66
7.5 A receive with the ref_trick. 66

List of Tables

12.1 List of memory allocators. 83

Preface

This book is not about how to write correct and beautiful code, I am assuming that you already know how to do that. This book isn't really about profiling and performance tuning either. Although, there is a chapter in this book on tracing and profiling which can help you find bottlenecks and unnecessary usage of resources. There also is a chapter on performance tuning.

These two chapters are the last chapters in the book, and the whole book is building up to those chapters, but the real goal with this book is to give you all the information, all the gory details, that you need in order to really understand the performance of your Erlang application.

About this book

For anyone who: Want to tune an Erlang installation. Want to know how to debug VM crashes. Want to improve performance of Erlang applications. Want to understand how Erlang really works. Want to learn how to build your own runtime environment.

If you want to debug the VM If you want to extend the VM If you want to do performance tweaking—jump to the last chapter … but to really understand that chapter you need to read the book.

How to read this book

The Erlang RunTime System (ERTS) is a complex system with many interdependent components. It is written in a very portable way so that it can run on anything from a gum-stick computer to the largest multicore system with terabytes of memory. In order to be able to optimize the performance of such a system for your application, you need to not only know your application, but you also need to have a thorough understanding of ERTS itself.

With this knowledge of how ERTS works you will be able to understand how your application behaves when running on ERTS, and you will also be able to find and fix problems with the performance of your application. In the second part of this book we will go through how you successfully run, monitor and scale your ERTS application.

You don't need to be an Erlang programmer to read this book, but you will need some basic understanding of what Erlang is. This following section will give you some Erlang background.

Erlang

In this section we will look at some basic Erlang concepts that are vital to understanding the rest of the book.

Erlang has been called, especially by one of Erlang's creators Joe Armstrong, a concurrency oriented language. Concurrency is definitely at the heart of Erlang, and to be able to understand how an Erlang system works you need to understand the concurrency model of Erlang.

First of all we need to make a distinction between *concurrency* and *parallelism*. In this book *concurrency* is the concept of having two or more processes that **can** execute independently of each other, this can be done by first executing one process then the other or by interleaving the execution, or by executing the processes in parallel. With *parallel* executions we mean that the processes actually execute at the exact same time by using several physical execution units. Parallelism can be achieved on different levels.

Through multiple execution units in the execution pipeline in one core, in several cores on one CPU, by several CPUs in one machine or through several machines.

Erlang uses processes to achieve concurrency. Conceptually Erlang processes are similar to most OS processes, they execute in parallel and can communicate through signals. In practices there is a huge difference in that Erlang processes are much more lightweight than most OS processes. Many other concurrent programming languages call their equivalent to Erlang processes *agents*.

Erlang achieves concurrency by interleaving the execution of processes on the Erlang virtual machine, the BEAM. On multi-core processor the BEAM can also achieve parallelism by running one scheduler per core and executing one Erlang process per scheduler. The designer of an Erlang system can achieve further parallelism by distributing the system on several computers.

A typical Erlang system (a server or service built in Erlang) consists of a number of Erlang *applications*, corresponding to a directory on disk. Each application is made up of several Erlang *modules* corresponding to files in the directory. Each module contains a number of *functions* and each function is made up of *expressions*.

Since Erlang is a functional language it has no statements, only expressions. Erlang expressions can be combined to an Erlang function. A function takes a number of arguments and returns a value. In Erlang Code Examples we can see some examples of Erlang expressions and functions.

Erlang Code Examples

```
%% Some Erlang expressions:

true.
1+1.
if (X > Y) -> X; true -> Y end.

%% An Erlang function:

max(X, Y) ->
  if (X > Y) -> X;
     true    -> Y
  end.
```

Erlang has a number of *built in functions* (or *BIFs*) which are implemented by the VM. This is either for efficiency reasons, like the implementation of `lists:append` (which could be implemented in Erlang). It could also be to provide some low level functionality which would be hard or impossible to implement in Erlang itself, like `list_to_atom`.

Since Erlang/OTP R13B03 you can also provide your own functions implemented in C by using the *Native Implemented Functions* (*NIF*) interface.

Part I

Understanding ERTS

Chapter 1

Introducing the Erlang Runtime System

The Erlang RunTime System (ERTS) is a complex system with many interdependent components. It is written in a very portable way so that it can run on anything from a gum stick computer to the largest multicore system with terabytes of memory. In order to be able to optimize the performance of such a system for your application, you need to not only know your application, but you also need to have a thorough understanding of ERTS itself.

1.1 ERTS and the Erlang Runtime System

There is a difference between any Erlang Runtime System and a specific implementation of an Erlang Runtime System. "Erlang/OTP" by Ericsson is the de facto standard implementation of Erlang and the Erlang Runtime System. In this book I will refer to this implementation as *ERTS* or spelled out *Erlang RunTime System* with a capital T. (See Section 1.3 for a definition of OTP.)

There is no official definition of what an Erlang Runtime System is, or what an Erlang Virtual Machine is. You could sort of imagine what such an ideal Platonic system would look like by taking ERTS and removing all the implementation specific details. This is unfortunately a circular definition, since you need to know the general definition to be able to identify an implementation specific detail. In the Erlang world we are usually to pragmatic to worry about this.

I will try to use the term *Erlang Runtime System* to refer to the general idea of any Erlang Runtime System as opposed to the specific implementation by Ericsson which I'll call the Erlang RunTime System or usually just ERTS.

Note This book is mostly a book about ERTS in particular and only to a small extent about any general Erlang Runtime System. If you assume that I talk about the Ericsson implementation unless I clearly state that I am talking about a general principle you will probably be right.

1.2 How to read this book

In Part II of this book I will show you how to tune the runtime system for your application and how to profile and debug your application and the runtime system. In order to really know how to tune the system you also need to know the system. In Part I of this book you will get a deep understanding of how the runtime system works.

In the following chapters of Part I I will try to explain each component of the system by itself, in one separate chapter for each of the major component. You should be able to read any one of these chapters without having a full understanding of how the other components are implemented, but you will need a basic understanding of what each component is. The rest of this introductory chapter should give you enough basic understanding and vocabulary to be able to jump between the rest of the chapters in part one in any order you like.

However, if you have the time I strongly recommend reading the book in order the first time. Words that are specific to Erlang and ERTS or used in a specific way in this book are usually explained at their first occurrence. Then, when you know the vocabulary, you can come back and use Part I as a reference whenever you have a problem with a particular component.

1.3 ERTS

In this section I will give a basic overview of the main components of ERTS and some vocabulary needed to understand the more detailed descriptions of each component in the following chapters.

1.3.1 The Erlang Node (ERTS)

An Erlang node is a running instance of ERTS (or possibly another implementation of Erlang (see Section 1.4)). In OO terminology one could say that an Erlang node is an object of the Erlang Runtime System class.

All execution of Erlang code is done within a node. An erlang node corresponds to an OS process and you can have several Erlang nodes running on one machine.

Your Erlang program (or application) will run in one or more Erlang nodes, and the performance of your program will depend not only on your application code but also on all the layers below your code in the *Erlang solution stack*. In particular if you are running your code on top of ERTS you will need to know the components of the *ERTS Stack*.

In [?] you can see the ERTS Stack illustrated with two Erlang nodes running on one machine.

In the bottom of the stack there is the hardware you are running on. The easiest way to improve the performance of your app is probably to run it on better hardware. If economical or physical constraints wont let you upgrade your hardware you can start exploring higher levels of the stack. The two most important choices for your hardware is whether it is multicore and whether it is 32-bit or 64-bit. You need different builds of ERTS depending on whether you want to use multicore or not and whether you want to use 32-bit or 64-bit. (See [?] for information on how to build different versions of ERTS.) This book will not go into any details about hardware but I will talk a bit about multicore and NUMA architectures in Chapter 11 and Chapter 12.

The second layer in the stack is the OS level. ERTS runs on most versions of Windows and most POSIX "compliant" OS:es, including Linux, VxWorks, Solaris, and Mac OS X. Today most of the development of ERTS is done on Linux and OS X, and you can expect the best performance on these platforms. However, Ericsson have been using Solaris internally in many projects and ERTS have been tuned for Solaris for many years. Depending on your use case you might actually get best performance on a Solaris system. The OS choice is usually not based on performance requirements, but is restricted by other demands. If you are building an embedded application you might be restricted to Rasbian or VxWork, and if you fore some reason are building an end user or client application you might have to use Windows. The Windows port of ERTS has so far not had the highest priority and might not be the best choice from a performance or maintenance perspective. If you want to use a 64-bit ERTS you of course need to have both a 64-bit machine and a 64-bit OS. I will not cover many OS specific questions in this book.

The third layer in the stack is the Erlang Runtime System. In our case this will be ERTS. This and the next layer, the Erlang Virtual Machine (BEAM), is what this book is all about. In the rest of Part I you will see how these layers work and are implemented, and in Part II you will see how you can tune ERTS to give your application optimal performance.

The fifth layer, OTP, supplies the Erlang standard libraries. OTP originally stood for "Open Telecom Platform" and was a number of Erlang libraries supplying building blocks (such as `supervisor`, `gen_server` and `gen_ftp`) for building robust applications (such as telephony exchanges). Early on, the libraries and the meaning of OTP got intermingled with all the other standard libraries shipped with ERTS. Nowadays most people use OTP together with Erlang in "Erlang/OTP" as the name for ERTS and all Erlang libraries shipped by Ericsson. Knowing these standard libraries and how and when to uses them can greatly improve the performance of your application. This book will not go into any details about the standard libraries and OTP, there are other books that cover these aspects.

Finally, the sixth layer (APP) is your application which can use all the functionality provided by the underlying layers. Apart from upgrading your hardware this is probably the place where you most easily can improve your application's performance. In [?] I will give some hints and show some tools that can help you profile and optimize your application. In [?] and [?] I'll give you some hints on hove to find the cause of crashing applications and how to find bugs in your application.

```
 Node1      Node2

+------+   +------+
| APP  |   | APP  |
+------+   +------+
| OTP  |   | OTP  |
+------+   +------+
| BEAM |   | BEAM |
```

```
+------+  +------+
| ERTS |  | ERTS |
+------+  +------+
+----------------+
|       OS       |
+----------------+
|    HW or VM    |
+----------------+
```

For information on how to build and run an Erlang node see [?], and read the rest of the book to learn all about the components of an Erlang node.

1.3.2 The Erlang Compiler

The Erlang Compiler is responsible for compiling Erlang source code, from .erl files into virtual machine code for BEAM (the virtual machine). The compiler itself is written in Erlang and compiled by itself to BEAM code and usually available in a running Erlang node. To bootstrap the runtime system there are a number of precompiled BEAM files, including the compiler, in the bootstrap directory.

For more information about the compiler see Chapter 2.

1.3.3 The Erlang Virtual Machine: BEAM

BEAM is the Erlang virtual machine used for executing Erlang code, just like the JVM is used for executing Java code. BEAM runs in an Erlang Node.

BEAM: The name BEAM originally stood for Bogdan's Erlang Abstract Machine, but now a days most people refer to it as Björn's Erlang Abstract machine, after the current maintainer.

Just as ERTS is an implementation of a more general concept of a Erlang Runtime System so is BEAM an implementation of a more general Erlang Virtual Machine (EVM) . There is no definition of what constitutes an EVM but BEAM actually has two levels of instructions *Generic Instructions* and *Specific Instructions*. The generic instruction set could be seen as a blueprint for an EVM.

For a full description of BEAM see Chapter 5, Chapter 6 and Chapter 7.

1.3.4 Processes

An Erlang process basically works like an OS process. Each process has its own memory (a mailbox, a heap and a stack) and a process control block (PCB) with information about the process.

All Erlang code execution is done within the context of a process. One Erlang node can have many processes, which can communicate through message passing and signals. Erlang processes can also communicate with processes on other Erlang nodes as long as the nodes are connected.

To learn more about processes and the PCB see Chapter 3.

1.3.5 Scheduling

The Scheduler is responsible for choosing the Erlang process to execute. Basically the scheduler keeps two queues, a *ready queue* of processes ready to run, and a *waiting queue* of processes waiting to receive a message. When a process in the waiting queue receives a message or get a time out it is moved to the ready queue.

The scheduler picks the first process from the ready queue and hands it to BEAM for execution of one *time slice*. BEAM preempts the running process when the time slice is used up and adds the processes to the end of the ready queue. If the process is blocked in a receive before the time slice is used up, it gets added to the waiting queue instead.

Erlang is concurrent by nature, that is, each process is conceptually running at the same time as all other processes, but in reality there is just one process running in the VM. On a multicore machine Erlang actually runs more than one scheduler, usually one per physical core, each having their own queues. This way Erlang achieves true parallelism. To utilize more than one core ERTS has to be built (see [?]) in *SMP* mode. SMP stands for *Symetric MultiProcessing*, that is, the ability to execute a processes on any one of multiple CPUs.

In reality the picture is more complicated with priorities among processes and the waiting queue is implemented through a timing wheel. All this and more is described in detail in Chapter 11.

1.3.6 The Erlang Tag Scheme

Erlang is a dynamically typed language, and the runtime system need a way to keep track of the type of each data object. This is done with a tagging scheme. Each data object or pointer to a data object also has a tag with information about the data type of the object.

Basically some bits of a pointer are reserved for the tag, and the emulator can then determine the type of the object by looking at the bit pattern of the tag.

These tags are used for pattern matching and for type test and for primitive operations as well as by the garbage collector.

The complete tagging scheme is described in Chapter 4.

1.3.7 Memory Handling

Erlang uses automatic memory management and the programmer does not have to worry about memory allocation and deallocation. Each process has a heap and a stack which both can grow, and shrink, as needed.

When a process runs out of heap space, the VM will first try to reclaim free heap space through garbage collection. The garbage collector will then go through the process stack and heap and copy live data to a new heap while throwing away all the data that is dead. If there still isn't enough heap space, a new larger heap will be allocated and the live data is moved there.

The details of the current generational copying garbage collector, including the handling of reference counted binaries can be found in Chapter 12.

In a system which uses HiPE compiled native code, each process actually has two stacks, a BEAM stack and a native stack, the details can be found in [?].

1.3.8 The Interpreter and the Command Line Interface

When you start an Erlang node with `erl` you get a command prompt. This is the *Erlang read eval print loop* (REPL) or the *command line interface* (CLI) or simply the *Erlang shell*.

You can actually type in Erlang code and execute it directly from the shell. In this case the code is not compiled to BEAM code and executed by the BEAM, instead the code is parsed and interpreted by the Erlang interpreter. In general the interpreted code behaves exactly as compiled code, but there a few subtle differences, these differences and all other aspects of the shell are explained in [?].

1.4 Other Erlang Implementations

This book is mainly concerned with the "standard" Erlang implementation by Ericsson/OTP called ERTS, but there are a few other implementations available and in this section I will discuss some of them briefly.

Throught the book I will sometimes mention differences between other implementations and ERTS, but there is no guarantee that I will mention all differences.

1.4.1 Erlang on Xen

Erlang on Xen (link:http://erlangonxen.org) is an Erlang implementation running directly on server hardware with no OS layer in between, only a thin Xen client.

Ling, the virtual machine of Erlang on Xen is almost 100% binary compatible with BEAM. In xref:the_eox_stack you can see how the Erlang on Xen implementation of the Erlang Solution Stack differs from the ERTS Stack. The thing to note here is that there is no operating system in the Erlang on Xen stack.

Since Ling implements the generic instruction set of BEAM, it can reuse the BEAM compiler from the OTP layer to compile Erlang to Ling.

```
 Node1     Node2       Node2     Node3

+------+  +------+   +------+  +------+
| APP  |  | APP  |   | APP  |  | APP  |
+------+  +------+   +------+  +------+
| OTP  |  | OTP  |   | OTP  |  | OTP  |
+------+  +------+   +------+  +------+
| Ling |  | Ling |   | BEAM |  | BEAM |
+------+  +------+   +------+  +------+
| EoX  |  | EoX  |   | ERTS |  | ERTS |
+------+  +------+   +------+  +------+
+----------------+   +----------------+
|      XEN       |   |       OS       |
+----------------+   +----------------+
|      HW        |   |    HW or VM    |
+----------------+   +----------------+
```

1.4.2 Erjang

Erjang (link:http://erjang.org) is an Erlang implementation which runs on the JVM. It loads `.beam` files and recompile the code to Java `.class` files. Erjang is almost 100% binary compatible with (generic) BEAM.

In xref:the_erjang_stack you can see how the Erjang implementation of the Erlang Solution Stack differs from the ERTS Stack. The thing to note here is that JVM has replaced BEAM as the virtual machine and that Erjang provides the services of ERTS by implementing them in Java on top of the VM.

```
 Node1     Node2       Node3     Node4

+------+  +------+   +------+  +------+
| APP  |  | APP  |   | APP  |  | APP  |
+------+  +------+   +------+  +------+
| OTP  |  | OTP  |   | OTP  |  | OTP  |
+------+  +------+   +------+  +------+
|Erjang|  |Erjang|   | BEAM |  | BEAM |
+------+  +------+   +------+  +------+
| JVM  |  | JVM  |   | ERTS |  | ERTS |
+------+  +------+   +------+  +------+
+----------------+   +----------------+
|      OS        |   |       OS       |
+----------------+   +----------------+
|  HW or VM      |   |    HW or VM    |
+----------------+   +----------------+
```

Now that you have a basic understanding of all the major pieces of ERTS, and the necessary vocabulary you can dive into the details of each component. If you are eager to understand a certain component, you can jump directly to that chapter. Or if you are really eager to find a solution to a specific problem you could jump to the right chapter in Part II, and try the different methods to tune, tweak, or debug your system. Although, I strongly suggest that you read through the chapters in Part I in order first in order to get a deep understanding of how ERTS really works.

Chapter 2

The Compiler

This book will not cover the programming language Erlang, but since the goal of the ERTS is to run Erlang code you will need to know how to compile Erlang code. In this chapter we will cover the compiler options needed to generate readable beam code and how to add debug information to the generated beam file. At the end of the chapter there is also a section on the Elixir compiler.

For those readers interested in compiling their own favorite language to ERTS this chapter will also contain detailed information about the different intermediate formats in the compiler and how to plug your compiler into the beam compiler backend. I will also present parse transforms and give examples of how to use them to tweak the Erlang language.

2.1 Compiling Erlang

Erlang is compiled from source code modules in `.erl` files to fat binary `.beam` files.

The compiler can be run from the OS shell with the `erlc` command:

```
> erlc foo.erl
```

Alternatively the compiler can be invoked from the Erlang shell with the default shell command `c` or by calling `compile:file/{1,2}`

```
1> c(foo).
```

or

```
1> compile:file(foo).
```

The optional second argument to `compile:file` is a list of compiler options. A full list of the options can be found in the documentation of the compile module: see http://www.erlang.org/doc/man/compile.html.

Normally the compiler will compile Erlang source code from a `.erl` file and write the resulting binary beam code to a `.beam` file. You can also get the resulting binary back as an Erlang term by giving the option `binary` to the compiler. This option has then been overloaded to mean return any intermediate format as a term instead of writing to a file. If you for example want the compiler to return Core Erlang code you can give the options `[core, binary]`.

The compiler is made up of a number of passes as illustrated in Compiler Passes Compiler Passes..

Compiler Passes.

```
[] = Compiler options, () = files, {} = erlang terms, boxes = passes </title>
          (.erl)
             |
             v
    +---------------+
    |    Scanner    |
```

```
    | (Part of epp) |
    +---------------+
            |
            v
    +---------------+
    | Pre-processor |
    |      epp      |
    +---------------+
            |
            v
    +---------------+    +---------------+
    |     Parse     | -> | user defined  |
    |   Transform   | <- | transformation|
    +---------------+    +---------------+
            |
            +---------> (.Pbeam) [makedep]
            +---------> {dep} [makedep, binary]
            |
            +---------> (.pp) [dpp]
            +---------> {AST} [dpp, binary]
            |
            v
    +---------------+
    |    Linter     |
    |               |
    +---------------+
            |
            +---------> (.P) ['P']
            +---------> {AST} ['P',binary]
            |
            v
    +---------------+
    |    Save AST   |
    |               |
    +---------------+
            |
            v
    +---------------+
    |     Expand    |
    |               |
    +---------------+
            |
            +---------> (.E) ['E']
            +---------> {.E} ['E', binary]
            |
            v
    +---------------+
    |     Core      |
    |    Erlang     |
    +---------------+
            |
            +---------> (.core) [dcore|to_core0]
            +---------> {core} [to_core0,binary]
            |
            v
    +---------------+
    |      Core     |+
    |     Passes    ||+
    +---------------+||
     +---------------+|
      +---------------+
            |
```

```
            +---------> (.core) [to_core]
            +---------> {core} [to_core,binary]
            |
            v
    +---------------+
    |    Kernel     |
    |    Erlang     |
    +---------------+
            |
            v
    +---------------+
    |    Kernel     |+
    |    Passes     ||+
    +---------------+||
     +---------------+|
      +---------------+
            |
            v
    +---------------+
    |   BEAM Code   |
    |               |
    +---------------+
            |
            v
    +---------------+
    |      ASM      |+
    |     Passes    ||+
    +---------------+||
     +---------------+|
      +---------------+
            |
            +---------> (.S) ['S']
            +---------> {.S} ['S', binary]
            |
            v
    +---------------+
    |  Native Code  |
    |               |
    +---------------+
            |
            v
         (.beam)</pre>
```

If you want to see a complete and up to date list of compiler passes you can run the function `compile:options/0` in an Erlang shell. The definitive source for information about the compiler is of course the source: compile.erl

2.2 Generating Intermediate Output

Looking at the code produced by the compiler is a great help in trying to understand how the virtual machine works. Fortunately, the compiler can show os the intermediate code after each compiler pass and the final beam code.

Let us try out our newfound knowledge to look at the generated code.

```
1> compile:options().
dpp - Generate .pp file
'P' - Generate .P source listing file
```

...

```
'E' - Generate .E source listing file
```

...

```
'S' - Generate .S file
```

Let us try with a small example program "world.erl":

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

-include("world.hrl").

hello() -> ?GREETING.
```

And the include file "world.hrl"

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

If you now compile this with the *P* option to get the parsed file you get a file "world.P":

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

In the resulting `.P` file you can see the a pretty printet version of the code after the preprocessor (and parse transformation) has been applied:

```
-file("world.erl", 1).

-module(world).

-export([hello/0]).

-file("world.hrl", 1).

-file("world.erl", 4).

hello() ->
    "hello world".
```

To see how the code looks after all source code transformations are done, you can compile the code with the `E`-flag.

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

This gives us an `.E` file, in this case all compiler directives have been removed and the build in functions `module_info/{1,2}` have been added to the source:

```
-vsn("\002").

-file("world.erl", 1).

-file("world.hrl", 1).

-file("world.erl", 5).

hello() ->
    "hello world".
```

```
module_info() ->
    erlang:get_module_info(world).

module_info(X) ->
    erlang:get_module_info(world, X).
```

We will make use of the *P* and *E* options when we look at parse transforms in Section 2.3.2, but first we will take a look at an "assembler" view of generated BEAM code. Bu giving the option *S* to the compiler you get a `.S` file with Erlang terms for each BEAM instruction in the code.

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

The file `world.S` should look like this:

```
{module, world}.  %% version = 0

{exports, [{hello,0},{module_info,0},{module_info,1}]}.

{attributes, []}.

{labels, 7}.


{function, hello, 0, 2}.
  {label,1}.
    {line,[{location,"world.erl",6}]}.
    {func_info,{atom,world},{atom,hello},0}.
  {label,2}.
    {move,{literal,"hello world"},{x,0}}.
    return.


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


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

Since this is a file with dot (".") separated Erlang terms, you can read the file back into the Erlang shell with:

```
{ok, BEAM_Code} = file:consult("world.S").
```

The assembler code mostly follows the layout of the original source code.

The first instruction defines the module name of the code. The version mentioned in the comment (`%% version =0`) is the version of the beam opcode format (as given by `beam_opcodes:format_number/0`).

Then comes a list of exports and any compiler attributes (none in this example) much like in any Erlang source module.

The first real beam-like instruction is `{labels, 7}` which tells the VM the number of lables in the code to make it possible to allocate room for all lables in one pass over the code.

After that there is the actual code for each function. The first instruction gives us the function name, the arity and the entry point as a label number.

You can use the *S* option with great effect to help you understand how the BEAM works, and we will use it like that in later chapters. It is also invaluable if you develop your own language that you compile to the BEAM through Core Erlang, to see the generated code.

2.3 Compiler Passes

In the following sections we will go through most of the compiler passes shown in Compiler Passes.. For a language designer targeting the BEAM this is interesting since it will show you what you can accomplish with the different approaches: macros, parse transforms, core erlang, and BEAM code, and how they depend on each other.

When tuning Erlang code, it is good to know what optimizations are applied when, and how you can look at generated code before and after optimizations.

2.3.1 Compiler Pass: The Erlang Preprocessor (epp)

The compilation starts with a combined tokenizer (or scanner) and preprocessor. That is, the preprosessor drives the tokenizer. This means that macros are expanded as tokens, so it is not a pure string replacement (as for example m4 or cpp). You can not use Erlang macros to define your own syntax, a macro will expand as a separate token from its surrounding characters. You can not concatenate a macro and a character to a token:

```
-define(plus,+).
t(A,B) -> A?plus+B.
```

This will expand to

```
t(A,B) -> A + + B.
```

and not

```
t(A,B) -> A ++ B.
```

On the other hand since macro expansion is done on the token level, you do not need to have a valid Erlang term in the right hand side of the macro, as long as you use it in a way that gives you a valid term. E.g.:

```
-define(p,o, o]).
 t() -> [f,?p.
```

There are few real useages for this other than to win the obfuscated Erlang code contest. The main point to remember is that you can not really use the Erlang preprocessor to define a language with a syntax that differs from Erlang. Fortunately there are other ways to do this, as you shall see later.

2.3.2 Compiler Pass: Parse Transformations

The easiest way to tweak the Erlang language is through Parse Transformations (or parse transforms). Parse Transformations comes with all sorts of warnings, like this note in the OTP documentation:

Warning
Programmers are strongly advised not to engage in parse transformations and no support is offered for problems encountered.

When you use a parse transform you are basically writing an extra pass in the compiler and that can if you are not careful lead to very unexpected results. But to use a parse transform you have to declare the usage in the module using it, and it will be local to that module, so as far as compiler tweaks goes this one is quite safe.

The biggest problem with parse transforms as I see it is that you are inventing your own syntax, and it will make it more difficult for anyone else reading your code. At least until your parse transform has become as popular and widely used as e.g. QLC.

OK, so you know you shouldn't use it, but if you have to, here is what you need to know. A parse transforms is a function that works on the abstract syntax tree (AST) (see http://www.erlang.org/doc/apps/erts/absform.html). The compiler does preprocessing, tokenization and parsing and then it will call the parse transform function with the AST and expects to get back a new AST.

This means that you can't change the Erlang syntax fundamentally, but you can change the semantics. Lets say for example that you for some reason would like to write json code directly in your Erlang code, then you are in luck since the tokens of json and of Erlang are basically the same. Also, since the Erlang compiler does most of its sanity checks in the linter pass which follows the parse transform pass, you can allow an AST which does not represent valid Erlang.

To write a parse transform you need to write an Erlang module (lets call it *p*) which exports the function `parse_transform/2`. This function is called by the compiler during the parse transform pass if the module being compiled (lets call it *m*) contains the compiler option `{parse_transform, p}`. The arguments to the funciton is the AST of the module m and the compiler options given to the call to the compiler.

Note
Note that you will not get any compiler options given in the file, this is a bit of a nuisance since you can't give options to the parse transform from the code.
The compiler does not expand compiler options until the *expand* pass which occures after the parse transform pass.

The documenation of the abstract format is somewhat dense and it is quite hard to get a grip on the abstract format by reading the documentation. I encourage you to use the *syntax_tools* and especially `erl_syntax_lib` for any serious work on the AST.

Here we will develop a a simple parse transform just to get an understanding of the AST. Therefore we will work directly on the AST and use the old reliable `io:format` approach instead of syntax_tools.

First we create an example of what we would like to be able to compile json_test.erl:

```
-module(json_test).
-compile({parse_transform, json_parser}).
-export([test/1]).

test(V) ->
    <<{{
      "name"  : "Jack (\"Bee\") Nimble",
      "format": {
                 "type"      : "rect",
                 "widths"     : [1920,1600],
                 "height"    : (-1080),
                 "interlace" : false,
                 "frame rate": V
                }
      }}>>.
```

Then we create a minimal parse transform module `json_parser.erl`:

```
-module(json_parser).
-export([parse_transform/2]).

parse_transform(AST, _Options) ->
  io:format("~p~n", [AST]),
  AST.
```

This identity parse transform returns an unchanged AST but it also prints it out so that you can see what an AST looks like.

```
> c(json_parser).
{ok,json_parser}
2> c(json_test).
[{attribute,1,file,{"./json_test.erl",1}},
 {attribute,1,module,json_test},
 {attribute,3,export,[{test,1}]},
 {function,5,test,1,
  [{clause,5,
    [{var,5,'V'}],
    [],
    [{bin,6,
      [{bin_element,6,
        {tuple,6,
         [{tuple,6,
           [{remote,7,{string,7,"name"},{string,7,"Jack (\"Bee\") Nimble"}},
            {remote,8,
             {string,8,"format"},
             {tuple,8,
              [{remote,9,{string,9,"type"},{string,9,"rect"}},
               {remote,10,
                {string,10,"widths"},
                {cons,10,
                 {integer,10,1920},
                 {cons,10,{integer,10,1600},{nil,10}}}},
               {remote,11,{string,11,"height"},{op,11,'-',{integer,11,1080}}},
               {remote,12,{string,12,"interlace"},{atom,12,false}},
               {remote,13,{string,13,"frame rate"},{var,13,'V'}}]}}]}]},
        default,default}]}]}]},
 {eof,16}]
./json_test.erl:7: illegal expression
./json_test.erl:8: illegal expression
./json_test.erl:5: Warning: variable 'V' is unused
error
```

The compilation of `json_test` fails since the module contains invalid Erlang syntax, but you get to see what the AST looks like. Now we can just write some functions to traverse the AST and rewrite the json code into Erlang code.[1]

```
-module(json_parser).
-export([parse_transform/2]).

parse_transform(AST, _Options) ->
    json(AST, []).

-define(FUNCTION(Clauses), {function, Label, Name, Arity, Clauses}).

%% We are only interested in code inside functions.
json([?FUNCTION(Clauses) | Elements], Res) ->
    json(Elements, [?FUNCTION(json_clauses(Clauses)) | Res]);
json([Other|Elements], Res) -> json(Elements, [Other | Res]);
json([], Res) -> lists:reverse(Res).
```

[1] The translation here is done in accordance with EEP 18 (Erlang Enhancement Proposal 18: "JSON bifs") link:http://www.erlang.org/eeps/eep-0018.html

```
%% We are interested in the code in the body of a function.
json_clauses([{clause, CLine, A1, A2, Code} | Clauses]) ->
    [{clause, CLine, A1, A2, json_code(Code)} | json_clauses(Clauses)];
json_clauses([]) -> [].


-define(JSON(Json), {bin, _, [{bin_element
                                        , _
                                        , {tuple, _, [Json]}
                                        , _
                                        , _}]}).

%% We look for: <<"json">> = Json-Term
json_code([])                     -> [];
json_code([?JSON(Json)|MoreCode]) -> [parse_json(Json) | json_code(MoreCode)];
json_code(Code)                   -> Code.

%% Json Object -> [{}] | [{Lable, Term}]
parse_json({tuple,Line,[]})            -> {cons, Line, {tuple, Line, []}};
parse_json({tuple,Line,Fields})        -> parse_json_fields(Fields,Line);
%% Json Array -> List
parse_json({cons, Line, Head, Tail})   -> {cons, Line, parse_json(Head),
                                                      parse_json(Tail)};
parse_json({nil, Line})                -> {nil, Line};
%% Json String -> <<String>>
parse_json({string, Line, String})     -> str_to_bin(String, Line);
%% Json Integer -> Intger
parse_json({integer, Line, Integer})   -> {integer, Line, Integer};
%% Json Float -> Float
parse_json({float, Line, Float})       -> {float, Line, Float};
%% Json Constant -> true | false | null
parse_json({atom, Line, true})         -> {atom, Line, true};
parse_json({atom, Line, false})        -> {atom, Line, false};
parse_json({atom, Line, null})         -> {atom, Line, null};

%% Variables, should contain Erlang encoded Json
parse_json({var, Line, Var})         -> {var, Line, Var};
%% Json Negative Integer or Float
parse_json({op, Line, '-', {Type, _, N}}) when Type =:= integer
                                              ; Type =:= float ->
                                          {Type, Line, -N}.
%% parse_json(Code)                  -> io:format("Code: ~p~n",[Code]), Code.

-define(FIELD(Lable, Code), {remote, L, {string, _, Label}, Code}).

parse_json_fields([], L) -> {nil, L};
%% Label : Json-Term  --> [{<<Label>>, Term} | Rest]
parse_json_fields([?FIELD(Lable, Code) | Rest], _) ->
    cons(tuple(str_to_bin(Label, L), parse_json(Code), L)
         , parse_json_fields(Rest, L)
         , L).


tuple(E1, E2, Line)    -> {tuple, Line, [E1, E2]}.
cons(Head, Tail, Line) -> {cons, Line, Head, Tail}.

str_to_bin(String, Line) ->
    {bin
     , Line
     , [{bin_element
         , Line
```

```
         , {string, Line, String}
         , default
         , default
        }
      ]
   }.
```

And now we can compile `json_test` without errors:

```
1> c(json_parser).
{ok,json_parser}
2> c(json_test).
{ok,json_test}
3> json_test:test(42).
[{<<"name">>,<<"Jack (\"Bee\") Nimble">>},
{<<"format">>,
  [{<<"type">>,<<"rect">>},
   {<<"widths">>,[1920,1600]},
   {<<"height">>,-1080},
   {<<"interlace">>,false},
   {<<"frame rate">>,42}]}]
```

The AST generated by `parse_teansfom/2` must correspond to valid Erlang code. Unless you apply several parse transforms, which is possible. The validity of the code is checked by the following compiler pass.

2.3.3 Compiler Pass: Linter

The linter (`erl_lint.erl`) generaters warnings for syntactically correct but otherwise bad code, like "export_all flag enabled".

2.3.4 Compiler Pass: Save AST

In order to enable debugging of a module, you can "debug compile" the module, that is pass the option `debug_info` to the compiler. The abstract syntax tree will then be saved by the "Save AST" until the end of the compilation, where it will be written to the .beam file.

It is important to note that the code is saved before any optimisations are appilied, so if there is a bug in an optimisation pass in the compiler and you run code in the debugger you will get a diffferent behavior. If you are implementing your own compiler optimisations this can trick you up badly.

2.3.5 Compiler Pass: Expand

In the expand phase source erlang constructs, such as records, are expanded to lower level erlang constructs. Compiler options, "`-compile(...)`", are also *expanded* to meta data.

2.3.6 Compiler Pass: Core Erlang

Core Erlang is a strict functional language suitable for compiler optimizations. It makes code transformations easier by reducing the number of ways to express the same operation. One way it does this is by introducing *let* and *letrec* expressions to make scoping more explicit.

Core Erlang is the best target for a language you want to run in ERTS. It changes very seldom and it contains all aspects of Erlang in a clean way. If you target the beam instruction set directly you will have to deal with much more detail, and that instruction set usually changes slightly between each major release of ERTS. If you on the other hand target Erlang directly you will be more restricted in what you can describe, and you will also have to deal with more details, since Core Erlang is a cleaner language.

To compile an Erlang file to core you can give the option "to_core", note though that this writes the Erlang core program to a file with the ".core" extension. To compile an Erlang core program from a ".core" file you can give the option "from_core" to the compiler.

```
1> c(world, to_core).
** Warning: No object file created - nothing loaded **
ok
2> c(world, from_core).
{ok,world}
```

Note that the `.core` files are text files written in the human readable core format. To get the core program as an Erlang term you can add the `binary` option to the compilation.

2.3.7 Compiler Pass: Kernel Erlang

Kernel Erlang is a flat version of Core Erlang with a few differences. For example, each variable is unique and the scope is a whole function. Pattern matching is compiled to more primitive operations.

2.3.8 Compiler Pass: BEAM Code

The last step of a normal compilation is the external beam code format. Some low level optimizations such as dead code elimination and peep hole optimisations are done on this level.

The BEAM code is described in detail in Chapter 7 and Appendix .1

2.3.9 Compiler Pass: Native Code

If you add the flag `native` to the compilation, and you have a HiPE enabled runtime system, then the compiler will generate native code for your module and store the native code along with the beam code in the `.beam.` file.

2.4 Other Compiler Tools

There are a number of tools available to help you work with code generation and code manipulation. These tools are written in Erlang and not really part of the runtime system but they are very nice to know about if you are implementing another language on top of the BEAM.

In this section we will cover three of the most useful code tools: the lexer — Leex, the parser generator — Yecc, and a general set of functions to manipulate abstract forms — Syntax Tools.

2.4.1 Leex

Leex is the Erlang lexer generator. The lexer generator takes a description of a DFA from a definitions file (<fileextension>xrl</fileextension>) and produces an Erlang program that matches tokens described by the DFA.

The details of how to write a DFA definition for a tokenizer is beyond the scope of this book. For a thorough explanation I recommend the "Dragon book" (Compiler ... by Aho, Sethi and Ullman). Other good resources are the man and info entry for "flex" the lexer program t inspired leex, and the leex documentation itself. If you have info and flex installed you can read the full manual by typing:

```
> info flex
```

The online Erlang documentation also has the leex manual (see [yecc.html](http://erlang.org/doc/man/yecc.html)).

We can use the lexer generator to create an Erlang program which recognizes JSON tokens. By looking at the JSON definition http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf we can see that there are only a handful of tokens that we need to handle.

```
Definitions.

Digit         = [0-9]
Digit1to9     = [1-9]
HexDigit      = [0-9a-f]
UnescapedChar = [^\"\\]
EscapedChar   = (\\\\)|(\\\")|(\\b)|(\\f)|(\\n)|(\\r)|(\\t)|(\\/)
Unicode       = (\\u{HexDigit}{HexDigit}{HexDigit}{HexDigit})
Quote         = [\"]
Delim         = [\[\]:,{}]
Space         = [\n\s\t\r]

Rules.

{Quote}{Quote} : {token, {string, TokenLine, ""}}.
{Quote}({EscapedChar}|({UnescapedChar})|({Unicode}))+{Quote} :
  {token, {string, TokenLine, drop_quotes(TokenChars)}}.

null  : {token, {null,  TokenLine}}.
true  : {token, {true,  TokenLine}}.
false : {token, {false, TokenLine}}.

{Delim} : {token, {list_to_atom(TokenChars), TokenLine}}.

{Space} : skip_token.

-?{Digit1to9}+{Digit}*\.{Digit}+((E|e)(\+|\-)?{Digit}+)? :
  {token, {number, TokenLine, list_to_float(TokenChars)}}.
-?{Digit1to9}+{Digit}* :
  {token, {number, TokenLine, list_to_integer(TokenChars)+0.0}}.

Erlang code.
-export([t/0]).

drop_quotes([$" | QuotedString]) -> literal(lists:droplast(QuotedString)).
literal([$\\,$" | Rest]) ->
  [$"|literal(Rest)];
literal([$\\,$\\ | Rest]) ->
  [$\\|literal(Rest)];
literal([$\\,$/ | Rest]) ->
  [$/|literal(Rest)];
literal([$\\,$b | Rest]) ->
  [$\b|literal(Rest)];
literal([$\\,$f | Rest]) ->
  [$\f|literal(Rest)];
literal([$\\,$n | Rest]) ->
  [$\n|literal(Rest)];
literal([$\\,$r | Rest]) ->
  [$\r|literal(Rest)];
literal([$\\,$t | Rest]) ->
  [$\t|literal(Rest)];
literal([$\\,$u,D0,D1,D2,D3|Rest]) ->
  Char = list_to_integer([D0,D1,D2,D3],16),
  [Char|literal(Rest)];
literal([C|Rest]) ->
  [C|literal(Rest)];
literal([]) ->[].

t() ->
  {ok,
   [{'{',1},
```

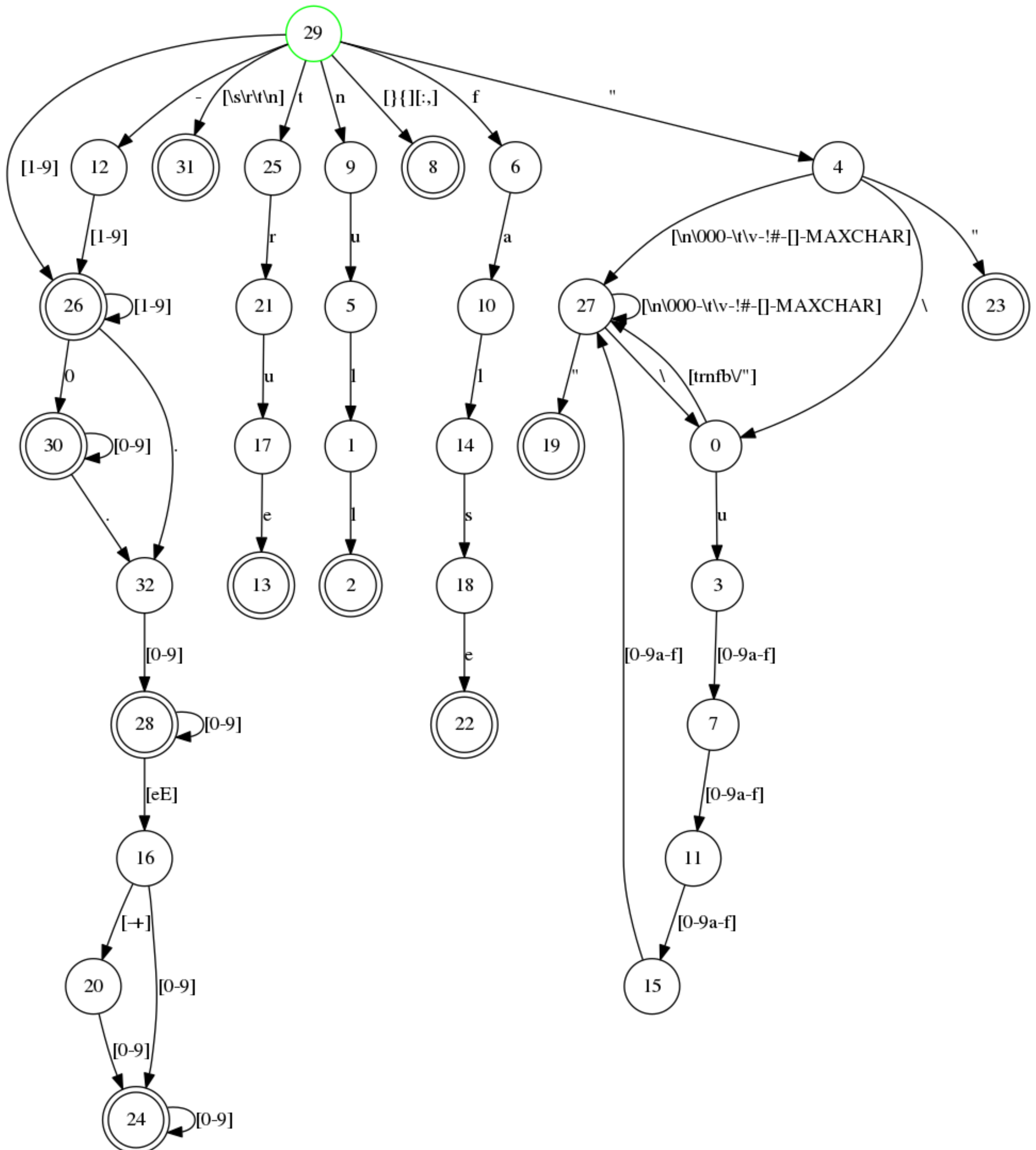
```
    {string,2,"no"},
    {':',2},
    {number,2,1.0},
    {'}',3}
   ],
   4}.
```

By using the Leex compiler we can compile this DFA to Erlang code, and by giving the option dfa_graph we also produce a dot-file which can be viewed with e.g. Graphviz.

```
1> leex:file(json_tokens, [dfa_graph]).
{ok, "./json_tokens.erl"}
2>
```

You can view the DFA graph using for example dotty.

```
> dotty json_tokens.dot
```



We can try our tokenizer on an example json file (test.json).

include::code/compiler_chapter/src/test.json

First we need to compile our tokenizer, then we read the file and convert it to a string. Finally we can use the string/1 function that leex generates to tokenize the test file.

```
2> c(json_tokens).
{ok,json_tokens}.
3> f(File), f(L), {ok, File} = file:read_file("test.json"), L = binary_to_lisile), ok.
```

```
ok
4> f(Tokens), {ok, Tokens,_} = json_tokens:string(L), hd(Tokens).
{'{',1}
5>
```

The shell function f/1 tells the shell to forget a variable binding. This is useful if you want to try a command that binds a variable multiple times, for example as you are writing the lexer and want to try it out after each rewrite. We will look at the shell commands in detail in the a later chapter.

Armed with a tokenizer for Json we can now write a json parser using the parser generator Yecc.

2.4.2 Yecc

Yecc is a parser generator for Erlang. The name comes from Yacc (Yet another compiler compiler) the canonical parser generator for C.

Now that we have a lexer for JSON terms we can write a parser using yecc.

Nonterminals value values object array pair pairs.

Terminals number string true false null *[ ] { } , :*.

Rootsymbol value.

value → object : *$1*. value → array : *$1*. value → number : get_val(*$1*). value → string : get_val(*$1*). value → *true* : get_val(*$1*). value → *null* : get_val(*$1*). value → *false* : get_val(*$1*).

object → *{ }* : #{}. object → *{* pairs *}* : *$2*.

pairs → pair : *$1*. pairs → pair *,* pairs : maps:merge(*$1*, *$3*).

pair → string *:* value : #{ get_val(*$1*) ⇒ *$3* }.

array → *[ ]* : {}. array → *[* values *]* : list_to_tuple(*$2*).

values → value : [*$1*]. values → value *,* values : [*$1* | *$3*].

Erlang code.

get_val({,,Val}) → Val; get_val({Val, _}) → Val.

Then we can use yecc to generate an Erlang program that implements the parser, and call the parse/1 function provided with the tokens generated by the tokenizer as argument.

```
5> yecc:file(yecc_json_parser), c(yecc_json_parser).
{ok,yexx_json_parser}
6> f(Json), {ok, Json} = yecc_json_parser:parse(Tokens).
{ok,#{"escapes" => "\b\n\r\t\f////",
      "format" => #{"frame rate" => 4.5,
        "height" => -1080.0,
        "interlace" => false,
        "type" => "rect",
        "unicode" => "/",
        "widths" => {1920.0,1.6e3}},
       "name" => "Jack \"Bee\" Nimble",
       "no" => 1.0}}
```

The tools Leex and Yecc are nice when you want to compile your own complete language to the Erlang virtual machine. By combining them with Syntax tools and specifically Merl you can manipulate the Erlang Abstract Syntax tree, either to generate Erlang code or to change the behaviour of Erlang code.

2.5 Syntax Tools and Merl

Syntax Tools is a set of libraries for manipulating the internal representation of Erlang's Abstract Syntax Trees (ASTs).

The syntax tools applications also includes the tool Merl since Erlang 18.0. With Merl you can very easily manipulate the syntax tree and write parse stransforms in Erlang code.

You can find the documentation for Syntax Tools on the Erlang.org site: [http://erlang.org/doc/apps/syntax_tools/chapter.html](http:/-/erlang.org/doc/apps/syntax_tools/chapter.html).

2.6 Compiling Elixir

Another approach to writing your own language on top of the Beam is to use the meta programming tools in Elixir. Elixir compiles to Beam code through the Erlang abstraxt syntax tree.

With Elixir's defmacro you can define your own Domain Specific Language, direcly in Elixir.

Chapter 3

Processes

The concept of lightweight processes is the essence of Erlang and the BEAM; they are what makes BEAM stand out from other virtual machines. In order to understand how the BEAM (and Erlang and Elixir) works you need to know the details of how processes work, which will help you understand the central concept of the BEAM, including what is easy and cheap for a process and what is hard and expensive.

Almost everything in the BEAM is connected to the concept of processes and in this chapter we will learn more about these connections. We will expand on what we learned in the introduction and take a deeper look at concepts such as memory management, message passing, and in particular scheduling.

An Erlang process is very similar to an OS process. It has its own address space, it can communicate with other processes through signals and messages, and the execution is controlled by a preemptive scheduler.

When you have a performance problem in an Erlang or Elixir system the problem is very often stemming from a problem within a particular process or from an imbalance between processes. There are of course other common problems such as bad algorithms or memory problems which we will look at in other chapters. Still, being able to pinpoint the process which is causing the problem is always important, therefore we will look at the tools available in the Erlang RunTime System for process inspection.

We will introduce the tools throughout the chapter as we go through how a process and the scheduler works, and then we will bring all tools together for an exercise at the end.

3.1 What is a Process?

A process is an isolated entity where code execution occurs. A process protects your system from errors in your code by isolating the effect of the error to the process executing the faulty code.

The runtime comes with a number of tools for inspecting processes to help us find bottlenecks, problems and overuse of resources. These tools will help you identify and inspect problematic processes.

3.1.1 Listing Processes from the Shell

Let us dive right in and look at which processes we have in a running system. The easiest way to do that is to just start an Erlang shell and issue the shell command `i()`. In Elixir you can call the function in the `shell_default` module as `:shell_default.i`.

```
$ erl
Erlang/OTP 19 [erts-8.1] [source] [64-bit] [smp:4:4] [async-threads:10]
              [hipe] [kernel-poll:false]

Eshell V8.1  (abort with ^G)
1> i().
Pid                     Initial Call                          Heap     Reds Msgs
```

```
Registered             Current Function                  Stack
<0.0.0>                otp_ring0:start/2                  376       579     0
init                   init:loop/1                          2
<0.1.0>                erts_code_purger:start/0           233         4     0
erts_code_purger       erts_code_purger:loop/0              3
<0.4.0>                erlang:apply/2                     987    100084     0
erl_prim_loader        erl_prim_loader:loop/3               5
<0.30.0>               gen_event:init_it/6                610       226     0
error_logger           gen_event:fetch_msg/5                8
<0.31.0>               erlang:apply/2                    1598       416     0
application_controlle gen_server:loop/6                     7
<0.33.0>               application_master:init/4          233        64     0
                       application_master:main_loop/2       6
<0.34.0>               application_master:start_it/4      233        59     0
                       application_master:loop_it/4         5
<0.35.0>               supervisor:kernel/1                610      1767     0
kernel_sup             gen_server:loop/6                    9
<0.36.0>               erlang:apply/2                    6772     73914     0
code_server            code_server:loop/1                   3
<0.38.0>               rpc:init/1                         233        21     0
rex                    gen_server:loop/6                    9
<0.39.0>               global:init/1                      233        44     0
global_name_server     gen_server:loop/6                    9
<0.40.0>               erlang:apply/2                     233        21     0
                       global:loop_the_locker/1             5
<0.41.0>               erlang:apply/2                     233         3     0
                       global:loop_the_registrar/0          2
<0.42.0>               inet_db:init/1                     233       209     0
inet_db                gen_server:loop/6                    9
<0.44.0>               global_group:init/1                233        55     0
global_group           gen_server:loop/6                    9
<0.45.0>               file_server:init/1                 233        79     0
file_server_2          gen_server:loop/6                    9
<0.46.0>               supervisor_bridge:standard_error/ 233        34     0
standard_error_sup     gen_server:loop/6                    9
<0.47.0>               erlang:apply/2                     233        10     0
standard_error         standard_error:server_loop/1         2
<0.48.0>               supervisor_bridge:user_sup/1       233        54     0
                       gen_server:loop/6                    9
<0.49.0>               user_drv:server/2                  987      1975     0
user_drv               user_drv:server_loop/6               9
<0.50.0>               group:server/3                     233        40     0
user                   group:server_loop/3                  4
<0.51.0>               group:server/3                     987     12508     0
                       group:server_loop/3                  4
<0.52.0>               erlang:apply/2                    4185      9537     0
                       shell:shell_rep/4                   17
<0.53.0>               kernel_config:init/1               233       255     0
                       gen_server:loop/6                    9
<0.54.0>               supervisor:kernel/1                233        56     0
kernel_safe_sup        gen_server:loop/6                    9
<0.58.0>               erlang:apply/2                    2586     18849     0
                       c:pinfo/1                           50
Total                                                   23426    220863     0
                                                          222
ok
```

The `i/0` function prints out a list of all processes in the system. Each process gets two lines of information. The first two lines of the printout are the headers telling you what the information means. As you can see you get the Process ID (Pid) and the name of the process if any, as well as information about the code the process is started with and is executing. You also get information about the heap and stack size and the number of reductions and messages in the process. In the rest of this chapter we will learn

in detail what a stack, a heap, a reduction and a message are. For now we can just assume that if there is a large number for the heap size, then the process uses a lot of memory and if there is a large number for the reductions then the process has executed a lot of code.

We can further examine a process with the `i/3` function. Let us take a look at the `code_server` process. We can see in the previous list that the process identifier (pid) of the `code_server` is `\<0.36.0\>`. By calling `i/3` with the three numbers of the pid we get this information:

```
2> i(0,36,0).
[{registered_name,code_server},
 {current_function,{code_server,loop,1}},
 {initial_call,{erlang,apply,2}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.35.0>]},
 {dictionary,[]},
 {trap_exit,true},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.33.0>},
 {total_heap_size,46422},
 {heap_size,46422},
 {stack_size,3},
 {reductions,93418},
 {garbage_collection,[{max_heap_size,#{error_logger => true,
                                       kill => true,
                                       size => 0}},
                      {min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,0}]},
 {suspending,[]}]
3>
```

We got a lot of information from this call and in the rest of this chapter we will learn in detail what most of these items mean. The first line tells us that the process has been given a name `code_server`. Next we can see which function the process is currently executing or suspended in (`current_function`) and the name of the function that the process started executing in (`initial_call`).

We can also see that the process is suspended waiting for messages (`{status,waiting}`) and that there are no messages in the mailbox (`{message_queue_len,0}`, `{messages,[]}`). We will look closer at how message passing works later in this chapter.

The fields `priority`, `suspending`, `reductions`, `links`, `trap_exit`, `error_handler`, and `group_leader` controls the process execution, error handling, and IO. We will look into this a bit more when we introduce the *Observer*.

The last few fields (`dictionary`, `total_heap_size`, `heap_size`, `stack_size`, and `garbage_collection`) gives us information about the process memory usage and we will look at the process memory areas in detail in this chapter in <ref linkend="ch.memory"/>.

Another, even more intrusive way of getting information about processes is to use the process information given by the `BREAK` menu: `ctrl+c p [enter]`. Note that while you are in the `BREAK` state the whole node freezes.

3.1.2 Programatic Process Probing

The shell functions just print the information about the process but you can actually get this information as data, so you can write your own tools for inspecting processes. You can get a list of all processes with `erlang:processes/0`, and more information about a process with `erlang:process_info/1`. We can also use the function `whereis/1` to get a pid from a name:

```
1> Ps = erlang:processes().
[<0.0.0>,<0.1.0>,<0.4.0>,<0.30.0>,<0.31.0>,<0.33.0>,
 <0.34.0>,<0.35.0>,<0.36.0>,<0.38.0>,<0.39.0>,<0.40.0>,
 <0.41.0>,<0.42.0>,<0.44.0>,<0.45.0>,<0.46.0>,<0.47.0>,
 <0.48.0>,<0.49.0>,<0.50.0>,<0.51.0>,<0.52.0>,<0.53.0>,
 <0.54.0>,<0.60.0>]
2> CodeServerPid = whereis(codeserver).
<0.36.0>
3> erlang:process_info(CodeServerPid).
[{registered_name,code_server},
 {current_function,{code_server,loop,1}},
 {initial_call,{erlang,apply,2}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.35.0>]},
 {dictionary,[]},
 {trap_exit,true},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.33.0>},
 {total_heap_size,24503},
 {heap_size,6772},
 {stack_size,3},
 {reductions,74260},
 {garbage_collection,[{max_heap_size,#{error_logger => true,
                                       kill => true,
                                       size => 0}},
                      {min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,33}]},
 {suspending,[]}]
```

By getting process information as data we can write code to analyze or sort the data as we please. If we grab all processes in the system (with `erlang:processes/0`) and then get information about the heap size of each process (with `erlang:process_info(P,total_heap_size)`) we can then construct a list with pid and heap size and sort it on heap size:

```
1> lists:reverse(lists:keysort(2,[{P,element(2,
    erlang:process_info(P,total_heap_size))}
    || P <- erlang:processes()])).
[{<0.36.0>,24503},
 {<0.52.0>,21916},
 {<0.4.0>,12556},
 {<0.58.0>,4184},
 {<0.51.0>,4184},
 {<0.31.0>,3196},
 {<0.49.0>,2586},
 {<0.35.0>,1597},
 {<0.30.0>,986},
 {<0.0.0>,752},
 {<0.33.0>,609},
 {<0.54.0>,233},
 {<0.53.0>,233},
 {<0.50.0>,233},
 {<0.48.0>,233},
 {<0.47.0>,233},
 {<0.46.0>,233},
 {<0.45.0>,233},
 {<0.44.0>,233},
 {<0.42.0>,233},
```

```
 {<0.41.0>,233},
 {<0.40.0>,233},
 {<0.39.0>,233},
 {<0.38.0>,233},
 {<0.34.0>,233},
 {<0.1.0>,233}]
2>
```

You might notice that many processes have a heap size of 233, that is because it is the default starting heap size of a process.

See the documentation of the module `erlang` for a full description of the information available with [`process_info`.](http://erlang.org/doc/man/erlang.html#process_info-1)

3.1.3 Using the Observer to Inspect Processes

A third way of examining processes is with the [*Observer*.](http://erlang.org/doc/apps/observer/observer_ug.html) The Observer is an extensive graphical interface for inspecting the Erlang RunTime System. We will use the Observer throughout this book to examine different aspects of the system.

The Observer can either be started from the OS shell and attach itself to an node or directly from an Elixir or Erlang shell. For now we will just start the Observer from the Elixir shell with `:observer.start` or from the Erlang shell with `observer:start()`.

When the Observer is started it will show you a system overview, see the following screen shot.

<figure id="fig-observer_system">

<imagedata fileref="images/observer_system.png"/> </figure> We will go over some of this information in detail later in this and the next chapter. For now we will just use the Observer to look at the running processes. First we take a look at the `Application` tab which shows the supervision tree of the running system:

`![](images/observer_applications.png)`

Here we get a graphical view of how the processes are linked. This is a very nice way to get an overview of how a system is structured. You also get a nice feeling of processes as isolated entities floating in space connected to each other through links.

To actually get some useful information about the processes we switch to the `Processes` tab:

`![](images/observer_processes.png)`

In this view we get basically the same information as with `i/0` in the shell. We see the pid, the registered name, number of reductions, memory usage and number of messages and the current function.

We can also look into a process by double clicking on its row, for example on the code server, to get the kind of information you can get with `process_info/2`:

`![](images/observer_code_server.png)`

We will not go through what all this information means right now, but if you keep on reading all will eventually be revealed.

Enabling the Observer [NOTE] If you are building your application with erlang.mk or rebar and you want to include the Observer application in your build you might need to add the applications `runtime_tools`, `wx`, and `observer` to your list of applications in yourapp.app.src.

Now that we have a basic understanding of what a process is and some tools to find and inspect processes in a system we are ready to dive deeper to learn how a process is implemented.

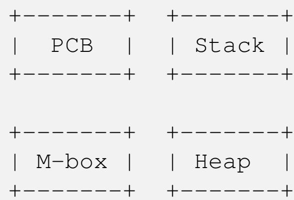
3.2 Processes Are Just Memory

A process is basically four blocks of memory: a *stack*, a *heap*, a *message area*, and the *Process Control Block* (*the PCB*).

The stack is used for keeping track of program execution by storing return addresses, for passing arguments to functions, and for keeping local variables. Larger structures, such as lists and tuples are stored on the heap.

The *message area*, also called *the mailbox*, is used to store messages sent to the process from other processes. The process control block is used to keep track of the state of the process.

See the following figure for an illustration of a process as memory:



```
+-------+  +-------+
|  PCB  |  | Stack |
+-------+  +-------+

+-------+  +-------+
| M-box |  | Heap  |
+-------+  +-------+
```

This picture of a process is very much simplified, and we will go through a number of iterations of more refined versions to get to a more accurate picture.

The stack, the heap, and the mailbox are all dynamically allocated and can grow and shrink as needed. We will see exactly how this works in later chapters. The PCB on the other hand is statically allocated and contains a number of fields that controls the process.

We can actually inspect some of these memory areas by using *HiPE's Built In Functions* (HiPE BIFs) for introspection. With these BIFs we can print out the memory content of stacks, heaps, and the PCB. The raw data is printed and in most cases a human readable version is pretty printed alongside the data. To really understand everything that we see when we inspect the memory we will need to know more about the Erlang tagging scheme (which we will go through in <ref linkend="ch.types"/> and about the execution model and error handling which we will go through in <ref linkend="ch.BEAM"/>), but using these tools will give us a nice view of how a process really is just memory.

HiPE's Built In Functions (HiPE BIFs)

The HiPE BIFs are not an official part of Erlang/OTP. They are not supported by the OTP team. They might be removed or changed at any time, so don't base your mission critical services on them.

These BIFs examines the internals of ERTS in a ways that might not be safe. The BIFs for introspection often just print to standard out and you might be surprised where that output ends up.

These BIFs can lock up a scheduler thread for a long time without using any reductions (we will look at what that means in the next chapter). Printing the heap of a very large process for example can take a long time.

Theses BIFs are only meant to be used for debugging and you use them at you own risk. You should probably not run them on a live system.

Many of the HiPE BIFs where written by the author in the mid nineties (before 64 bits Erlang existed) and the printouts on a 64 bit machine might be a bit off. There are new versions of these BIFs that do a better job, hopefully they will be included in ERTS at the time of the printing of this book. Otherwise you can build your own version with the patch provided in the code section and the instructions in <ref linkend="ap.building_erts"/>.

We can see the context of the stack of a process with `hipe_bifs:show_estack/1`:

```
1> hipe_bifs:show_estack(self()).
 |                BEAM  STACK              |
 |            Address |           Contents |
 |--------------------|--------------------| BEAM ACTIVATION RECORD
 | 0x00007f9cc3238310 | 0x00007f9cc2ea6fe8 | BEAM PC shell:exprs/7 + 0x4e
 | 0x00007f9cc3238318 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238320 | 0x000000000000644b | none
 |--------------------|--------------------| BEAM ACTIVATION RECORD
 | 0x00007f9cc3238328 | 0x00007f9cc2ea6708 | BEAM PC shell:eval_exprs/7 + 0xf
 | 0x00007f9cc3238330 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238338 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238340 | 0x000000000004f3cb | cmd
 | 0x00007f9cc3238348 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238350 | 0x00007f9cc3237102 | {value,#Fun<shell.5.104321512>}
 | 0x00007f9cc3238358 | 0x00007f9cc323711a | {eval,#Fun<shell.21.104321512>}
```

```
 | 0x00007f9cc3238360 | 0x00000000000200ff | 8207
 | 0x00007f9cc3238368 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238370 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238378 | 0xfffffffffffffffb | []
 |--------------------|--------------------| BEAM ACTIVATION RECORD
 | 0x00007f9cc3238380 | 0x00007f9cc2ea6300 | BEAM PC shell:eval_loop/3 + 0x47
 | 0x00007f9cc3238388 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238390 | 0xfffffffffffffffb | []
 | 0x00007f9cc3238398 | 0xfffffffffffffffb | []
 | 0x00007f9cc32383a0 | 0xfffffffffffffffb | []
 | 0x00007f9cc32383a8 | 0x000001a000000343 | <0.52.0>
 |....................|....................| BEAM CATCH FRAME
 | 0x00007f9cc32383b0 | 0x0000000000005a9b | CATCH 0x00007f9cc2ea67d8
 |                    |                    |  (BEAM shell:eval_exprs/7 + 0x29)
 |********************|********************|
 |--------------------|--------------------| BEAM ACTIVATION RECORD
 | 0x00007f9cc32383b8 | 0x000000000093aeb8 | BEAM PC normal-process-exit
 | 0x00007f9cc32383c0 | 0x00000000000200ff | 8207
 | 0x00007f9cc32383c8 | 0x000001a000000343 | <0.52.0>
 |--------------------|--------------------|
true
2>
```

We will look closer at the values on the stack and the heap in <ref linkend="ch.types"/>. The content of the heap is printed by `hipe_bifs:show_heap/1`. Since we do not want to list a large heap here we'll just spawn a new process that does nothing and show that heap:

```
2> hipe_bifs:show_heap(spawn(fun () -> ok end)).
From: 0x00007f7f33ec9588 to 0x00007f7f33ec9848
 |                 H E A P                 |
 |            Address |           Contents |
 |--------------------|--------------------|
 | 0x00007f7f33ec9588 | 0x00007f7f33ec959a | #Fun<erl_eval.20.52032458>
 | 0x00007f7f33ec9590 | 0x00007f7f33ec9839 | [[]]
 | 0x00007f7f33ec9598 | 0x0000000000000154 | Thing Arity(5) Tag(20)
 | 0x00007f7f33ec95a0 | 0x00007f7f3d3833d0 | THING
 | 0x00007f7f33ec95a8 | 0x0000000000000000 | THING
 | 0x00007f7f33ec95b0 | 0x0000000000600324 | THING
 | 0x00007f7f33ec95b8 | 0x0000000000000000 | THING
 | 0x00007f7f33ec95c0 | 0x0000000000000001 | THING
 | 0x00007f7f33ec95c8 | 0x000001d0000003a3 | <0.58.0>
 | 0x00007f7f33ec95d0 | 0x00007f7f33ec95da | {[],{eval...
 | 0x00007f7f33ec95d8 | 0x0000000000000100 | Arity(4)
 | 0x00007f7f33ec95e0 | 0xfffffffffffffffb | []
 | 0x00007f7f33ec95e8 | 0x00007f7f33ec9602 | {eval,#Fun<shell.21.104321512>}
 | 0x00007f7f33ec95f0 | 0x00007f7f33ec961a | {value,#Fun<shell.5.104321512>}...
 | 0x00007f7f33ec95f8 | 0x00007f7f33ec9631 | [{clause...

 ...

 | 0x00007f7f33ec97d0 | 0x00007f7f33ec97fa | #Fun<shell.5.104321512>
 | 0x00007f7f33ec97d8 | 0x00000000000000c0 | Arity(3)
 | 0x00007f7f33ec97e0 | 0x0000000000000e4b | atom
 | 0x00007f7f33ec97e8 | 0x000000000000001f | 1
 | 0x00007f7f33ec97f0 | 0x0000000000006d0b | ok
 | 0x00007f7f33ec97f8 | 0x0000000000000154 | Thing Arity(5) Tag(20)
 | 0x00007f7f33ec9800 | 0x00007f7f33bde0c8 | THING
 | 0x00007f7f33ec9808 | 0x00007f7f33ec9780 | THING
 | 0x00007f7f33ec9810 | 0x000000000060030c | THING
 | 0x00007f7f33ec9818 | 0x0000000000000002 | THING
 | 0x00007f7f33ec9820 | 0x0000000000000001 | THING
 | 0x00007f7f33ec9828 | 0x000001d0000003a3 | <0.58.0>
```

```
 | 0x00007f7f33ec9830 | 0x000001a000000343 | <0.52.0>
 | 0x00007f7f33ec9838 | 0xfffffffffffffffb | []
 | 0x00007f7f33ec9840 | 0xfffffffffffffffb | []
 |--------------------|--------------------|
true
3>
```

We can also print the content of some of the fields in the PCB with ‘hipe_bifs:show_pcb/1’:

```
3> hipe_bifs:show_pcb(self()).
P: 0x00007f7f3cbc0400
```

Offset| Name | Value | *Value | 0 | id | 0x000001d0000003a3 | | 72 | htop | 0x00007f7f33f15298 | | 96 | hend | 0x00007f7f33f16540 | | 88 | heap | 0x00007f7f33f11470 | | 104 | heap_sz | 0x0000000000000a1a | | 80 | stop | 0x00007f7f33f16480 | | 592 | gen_gcs | 0x0000000000000012 | | 594 | max_gen_gcs | 0x000000000000ffff | | 552 | high_water | 0x00007f7f33f11c50 | | 560 | old_hend | 0x00007f7f33e90648 | | 568 | old_htop | 0x00007f7f33e8f8e8 | | 576 | old_head | 0x00007f7f33e8e770 | | 112 | min_heap_.. | 0x00000000000000e9 | | 328 | rcount | 0x0000000000000000 | | 336 | reds | 0x0000000000002270 | | 16 | tracer | 0xfffffffffffffffb | | 24 | trace_fla.. | 0x0000000000000000 | | 344 | group_lea.. | 0x0000019800000333 | | 352 | flags | 0x0000000000002000 | | 360 | fvalue | 0xfffffffffffffffb | | 368 | freason | 0x0000000000000000 | | 320 | fcalls | 0x00000000000005a2 | | 384 | next | 0x0000000000000000 | | 48 | reg | 0x0000000000000000 | | 56 | nlinks | 0x00007f7f3cbc0750 | | 616 | mbuf | 0x0000000000000000 | | 640 | mbuf_sz | 0x0000000000000000 | | 464 | dictionary | 0x0000000000000000 | | 472 | seq..clock | 0x0000000000000000 | | 480 | seq..astcnt | 0x0000000000000000 | | 488 | seq..token | 0xfffffffffffffffb | | 496 | intial[0] | 0x000000000000320b | | 504 | intial[1] | 0x0000000000000c8b | | 512 | intial[2] | 0x0000000000000002 | | 520 | current | 0x00007f7f3be87c20 | 0x000000000000ed8b | 296 | cp | 0x00007f7f3d3a5100 | 0x0000000000440848 | 304 | i | 0x00007f7f3be87c38 | 0x000000000044353a | 312 | catches | 0x0000000000000001 | | 224 | arity | 0x0000000000000000 | | 232 | arg_reg | 0x00007f7f3cbc04f8 | 0x000000000000320b | 240 | max_arg_reg | 0x0000000000000006 | | 248 | def..reg[0] | 0x000000000000320b | | 256 | def..reg[1] | 0x0000000000000c8b | | 264 | def..reg[2] | 0x00007f7f33ec9589 | | 272 | def..reg[3] | 0x0000000000000000 | | 280 | def..reg[4] | 0x0000000000000000 | | 288 | def..reg[5] | 0x00000000000007d0 | | 136 | nsp | 0x0000000000000000 | | 144 | nstack | 0x0000000000000000 | | 152 | nstend | 0x0000000000000000 | | 160 | ncallee | 0x0000000000000000 | | 56 | ncsp | 0x0000000000000000 | | 64 | narity | 0x0000000000000000 | |

```
true
4>
```

Now armed with these inspection tools we are ready to look at what this fields in the PCB means.

3.3 The PCB

The Process Control Block contains all the fields that control the behavior and current state of a process. In this section and the rest of the chapter we will go through the most important fields. We will leave out some fields that have to do with execution and tracing from this chapter, instead we will cover those in <ref linkend="ch.BEAM"/>.

If you want to dig even deeper than we will go in this chapter you can look at the C source code. The PCB is implemented as a C struct called `process` in the file [`erl_process.h`.](https://github.com/erlang/otp/blob/OTP_R16B03-1/erts/emulator/-beam/erl_process.h)

The field `id` contains the process ID (or PID).

```
    0 | id          | 0x000001d0000003a3 |                    |
```

The process id is an Erlang term and hence tagged. (See <ref linkend="sec.tagging"/>) This means that the 4 least significant bits are a tag (0011). In the code section there is a module for inspecting Erlang terms (<filename>show.erl</filename>) which we will cover in the chapter on types. We can use it now to to examine the type of a tagged word though.

```
4> show:tag_to_type(16#0000001d0000003a3).
pid
5>
```

The fields `htop` and `stop` are pointers to the top of the heap and the stack, that is, they are pointing to the next free slots on the heap or stack. The fields `heap` (start) and `hend` points to the start and the stop of the whole heap, and `heap_sz` gives the size of the heap in words. That is `hend -heap =heap_sz * 8` on a 64 bit machine and `hend -heap =heap_sz * 4` on a 32 bit machine.

The field `heap_min_size` is the size, in words, that the heap starts with and which it will not shrink smaller than, the default value is 233.

We can now refine the picture of the process heap with the fields from the PCB that controls the shape of the heap:

```
hend ->  +----+    -
         |    |    ^
         |    |    |             -
htop ->  |    |    | heap_sz*8   ^
         |....|    | hend-heap   | min_heap_size
         |....|    v             v
heap ->  +----+    -             -
        The Heap
```

But wait, how come we have a heap start and a heap end, but no start and stop for the stack? That is because the BEAM uses a trick to save space and pointers by allocating the heap and the stack together. It is time for our first revision of our process as memory picture. The heap and the stack are actually just one memory area:

```
+-------+  +-------+
|  PCB  |  | Stack |
+-------+  +-------+
           | free  |
+-------+  +-------+
| M-box |  | Heap  |
+-------+  +-------+
```

The stack grows towards lower memory addresses and the heap towards higher memory, so we can also refine the picture of the heap by adding the stack top pointer to the picture:

```
hend ->  +----+    -
         |....|    ^
stop ->  |    |    |
         |    |    |
         |    |    |             -
htop ->  |    |    | heap_sz     ^
         |....|    |             | min_heap_size
         |....|    v             v
heap ->  +----+    -             -
        The Heap
```

If the pointers `htop` and `stop` were to meet, the process would run out of free memory and would have to do a garbage collection to free up memory.

3.4 The Garbage Collector (GC)

The heap memory management schema is to use a per process copying generational garbage collector. When there is no more space on the heap (or the stack, since they share the allocated memory block), the garbage collector kicks in to free up memory.

The GC allocates a new memory area called *to space*. Then it goes through the stack to find all live roots and follows each root and copies the data on the heap to the new heap. Finally it also copies the stack to the new heap and frees up the old memory area.

The GC is controlled by these fields int the PCB:

```
    Eterm *high_water;
    Eterm *old_hend;    /* Heap pointers for generational GC. */
    Eterm *old_htop;
    Eterm *old_heap;
    Uint max_heap_size; /* Maximum size of heap (in words). */
    Uint16 gen_gcs;     /* Number of (minor) generational GCs. */
    Uint16 max_gen_gcs; /* Max minor gen GCs before fullsweep. */
```

Since the garbage collector is generational it will use a heuristics to just look at new data most of the time. That is, in what is called a *minor collection*, the GC only looks at the top part of the stack and moves new data to the new heap. Old data, that is data allocated below the `high_water` mark (see the figure below) on the heap, is moved to a special area called the old heap.

Most of the time, then, there is another heap area for each process: the old heap, handled by the fields `old_heap`, `old_htop` and `old_hend` in the PCB. This almost bings us back to our original picture of a process as four memory areas:

```
  +-------+                +-------+
  |  PCB  |                | Stack |   +-------+  - old_hend
  +-------+                +-------+   +       +  - old_htop
                           | free  |   +-------+
  +-------+ high_water ->  +-------+   |  Old  |
  | M-box |                | Heap  |   | Heap  |
  +-------+                +-------+   +-------+  - old_heap
```

When a process starts there is no old heap, but as soon as young data has matured to old data and there is a garbage collection, the old heap is allocated. The old heap is garbage collected when there is a *major collection*, also called a *full sweep*. See <ref linkend="ch.memory"/> for more details of how garbage collection works. In that chapter we will also look at how to track down and fix memory related problems.

3.5 Mailboxes and Message Passing

Process communication is done through message passing. A process send is implemented so that a sending process copies the message from its own heap to the mailbox of the receiving process.

In the early days of Erlang concurrency was implemented through multitasking in the scheduler. We will talk more about concurrency in the section about the scheduler later in this chapter, for now it is worth noting that in the first version of Erlang there was no parallelism and there could only be one process running at the time. In that version the sending process could write data directly on the receiving process' heap.

3.5.1 Sending Messages in Parallel

When multicore systems were introduced and the Erlang implementation was extended with several schedulers running processes in parallel it was no longer safe to write directly on another process' heap without taking the `main lock` of the receiver. At this time the concept of `m-bufs` was introduced (also called `heap fragments`). An `m-buf` is a memory area outside of a process heap where other processes can safely write data. If a sending process can not get the lock it would write to the `m-buf` instead. When all data of a message has been copied to the `m-buf` the message is linked to the process through the mailbox. The linking (*LINK_MESSAGE* in <filename>erl_message.h</filename>) appends the message to the receiver's message queue.

The garbage collector would then copy the messages onto the process' heap. To reduce the pressure on the GC the mailbox is divided into two lists, one containing seen messages and one containing new messages. The GC does not have to look at the new messages since we know they will survive (they are still in the mailbox) and that way we can avoid some copying.

3.6 Lock Free Message Passing

In Erlang 19 a new per process setting was introduced, *message_queue_data*, which can take the values *on_heap* or *off_heap*. When set to *on_heap* the sending process will first try to take the `main lock` of the receiver and if it succeeds the message

will be copied directly onto the receiver's heap. This can only be done if the receiver is suspended and if no other process has grabbed the lock to send to the same process. If the sender can not obtain the lock it will allocate a heap fragment and copy the message there instead.

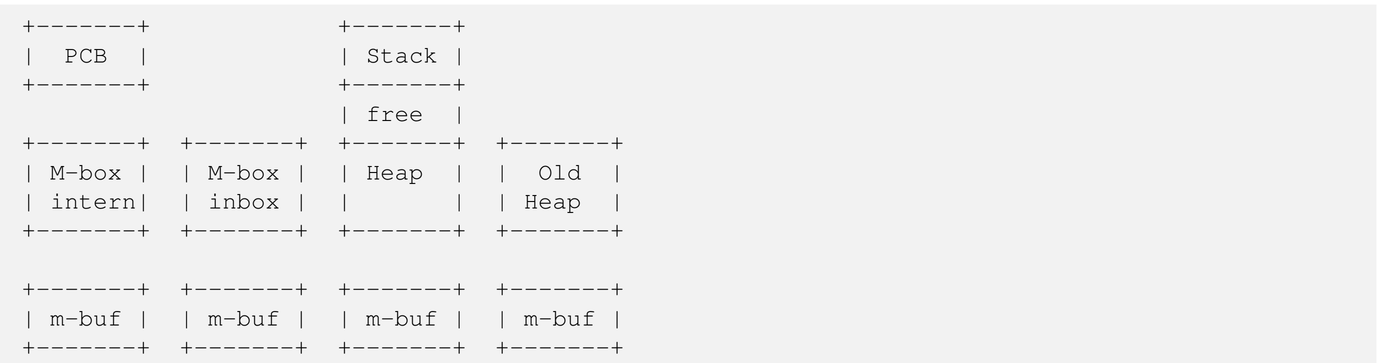
If the flag is set to *off_heap* the sender will not try to get the lock and instead write directly to a heap fragment. This will reduce lock contention but allocating a heap fragment is more expensive than writing directly to the already allocated process heap and it can lead to larger memory usage. There might be a large empty heap allocated and still new messages are written to new fragments.

With *on_heap* allocation all the messages, both directly allocated on the heap and messages in heap fragments, will be copied by the GC. If the message queue is large and many messages are not handled and therefore still are live, they will be promoted to the old heap and the size of the process heap will increase, leading to higher memory usage.

All messages are added to a linked list (the mailbox) when the message has been copied to the receiving process. If the message is copied to the heap of the receiving process the message is linked in to the `internal message queue` (or `seen` messages) and examined by the GC. In the *off_heap* allocation scheme new messages are placed in the "external" `message in queue` and ignored by the GC.

3.6.1 Memory Areas for Messages

We can now revise our picture of the process as four memory areas once more. Now the process is made up of five memory areas (two mailboxes) and a varying number of heap fragments (`m-bufs`):



```
+-------+             +-------+
|  PCB  |             | Stack |
+-------+             +-------+
                      | free  |
+-------+  +-------+  +-------+  +-------+
| M-box |  | M-box |  | Heap  |  |  Old  |
| intern|  | inbox |  |       |  | Heap  |
+-------+  +-------+  +-------+  +-------+

+-------+  +-------+  +-------+  +-------+
| m-buf |  | m-buf |  | m-buf |  | m-buf |
+-------+  +-------+  +-------+  +-------+
```

Each mailbox consists of a length and two pointers, stored in the fields *msg.len*, *msg.first*, *msg.last* for the internal queue and *msg_inq.len*, *msg_inq.first*, and `msg_inq.last` for the external in queue. There is also a pointer to the next message to look at (`msg.save`) to implement selective receive.

3.6.2 Inspecting Message Handling

Let us use our introspection tools to see how this works in more detail. We start by setting up a process with a message in the mailbox and then take a look at the PCB.

```
4> P = spawn(fun() -> receive stop -> ok end end).
<0.63.0>
5> P ! start.
start
6> hipe_bifs:show_pcb(P).

...
  408 | msg.first     | 0x00007fd40962d880 |                   |
  416 | msg.last      | 0x00007fd40962d880 |                   |
  424 | msg.save      | 0x00007fd40962d880 |                   |
  432 | msg.len       | 0x0000000000000001 |                   |
  696 | msg_inq.first | 0x0000000000000000 |                   |
  704 | msg_inq.last  | 0x00007fd40a306238 |                   |
  712 | msg_inq.len   | 0x0000000000000000 |                   |
  616 | mbuf          | 0x0000000000000000 |                   |
```

```
  640 | mbuf_sz       | 0x0000000000000000 |                     |
...
```

From this we can see that there is one message in the message queue and the `first`, `last` and `save` pointers all points to this message.

As mentioned we can force the message to end up in the in queue by setting the flag *message_queue_data*. We can try this with the following program:

<embed file= "code/msg.erl"/>

With this program we can try sending a message on heap and off heap and look at the PCB after each send. With on heap we get the same result as when just sending a message before:

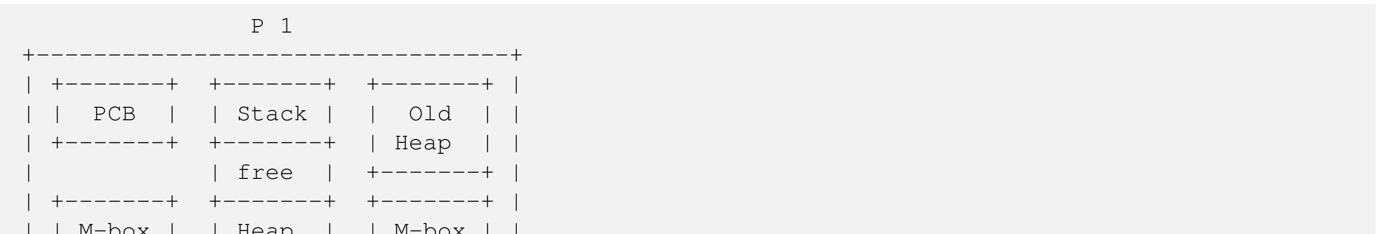
```
5> msg:send_on_heap().

...

  408 | msg.first     | 0x00007fd4096283c0 |                     |
  416 | msg.last      | 0x00007fd4096283c0 |                     |
  424 | msg.save      | 0x00007fd40a3c1048 |                     |
  432 | msg.len       | 0x0000000000000001 |                     |
  696 | msg_inq.first | 0x0000000000000000 |                     |
  704 | msg_inq.last  | 0x00007fd40a3c1168 |                     |
  712 | msg_inq.len   | 0x0000000000000000 |                     |
  616 | mbuf          | 0x0000000000000000 |                     |
  640 | mbuf_sz       | 0x0000000000000000 |                     |

...
```

If we try sending to a process with the flag set to *off_heap* the message ends up in the in queue instead:

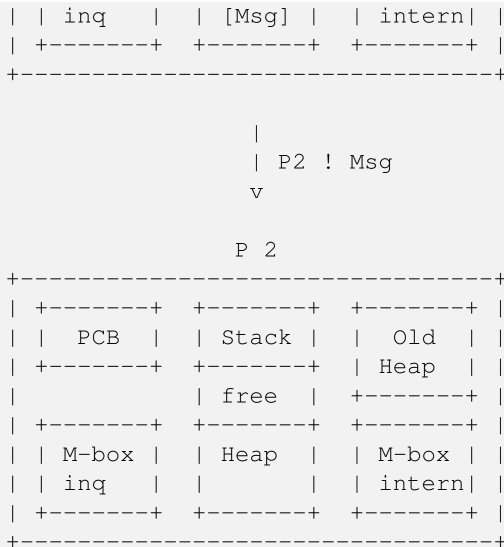
```
6> msg:send_off_heap().

...

  408 | msg.first     | 0x0000000000000000 |                     |
  416 | msg.last      | 0x00007fd40a3c0618 |                     |
  424 | msg.save      | 0x00007fd40a3c0618 |                     |
  432 | msg.len       | 0x0000000000000000 |                     |
  696 | msg_inq.first | 0x00007fd3b19f1830 |                     |
  704 | msg_inq.last  | 0x00007fd3b19f1830 |                     |
  712 | msg_inq.len   | 0x0000000000000001 |                     |
  616 | mbuf          | 0x0000000000000000 |                     |
  640 | mbuf_sz       | 0x0000000000000000 |                     |

...
```

3.6.3 The Process of Sending a Message to a Process

We will ignore the distribution case for now, that is we will not consider messages sent between Erlang nodes. Imagine two processes `P1` and `P2`. Process `P1` wants to send a message (*Msg*) to process `P2`, as illustrated by this figure:



```
                 P 1
 +---------------------------------+
 | +-------+  +-------+  +-------+ |
 | |  PCB  |  | Stack |  |  Old  | |
 | +-------+  +-------+  | Heap  | |
 |            | free  |  +-------+ |
 | +-------+  +-------+  +-------+ |
 | | M-box |  | Heap  |  | M-box | |
```

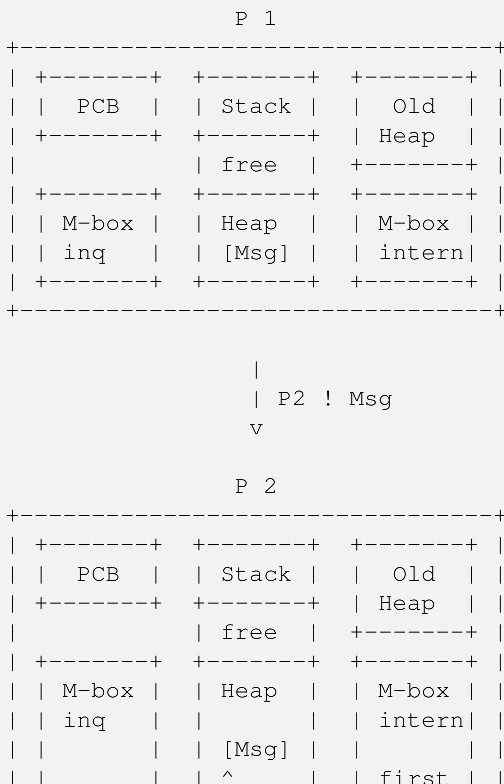


```
| | inq    |  | [Msg] |  | intern| |
| +-------+  +-------+  +-------+ |
+-----------------------------------+

                  |
                  | P2 ! Msg
                  v

                P 2
+-----------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  |       |  | intern| |
| +-------+  +-------+  +-------+ |
+-----------------------------------+
```

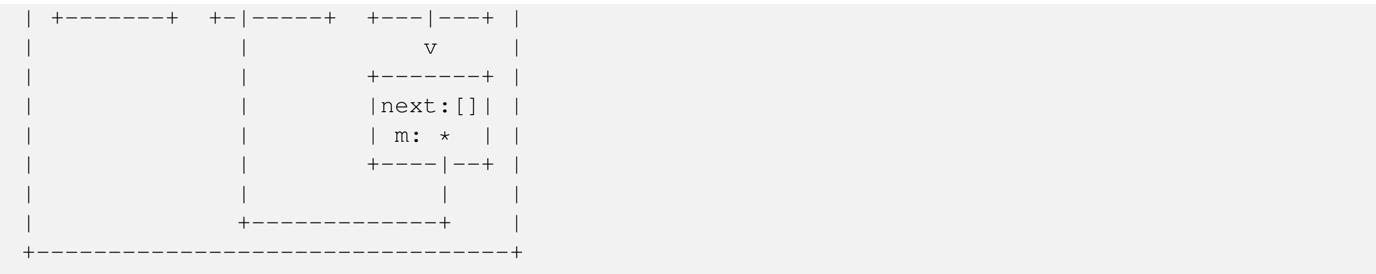
Process `P1` will then take the following steps:

- Calculate the size of *Msg*.
- Allocate space for the message (on or off `P2's` heap as described before).
- Copy *Msg* from `P1's` heap to the allocated space.
- Allocate and fill in an *ErlMessage* struct wrapping up the message.
- Link in the *ErlMessage* either in the *ErlMsgQueue* or in the *ErlMsgInQueue*.

If process `P2` is suspended and no other process is trying to send a message to `P2` and there is space on the heap and the allocation strategy is *on_heap* the message will directly end up on the heap:



```
                P 1
+-----------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  | [Msg] |  | intern| |
| +-------+  +-------+  +-------+ |
+-----------------------------------+

                  |
                  | P2 ! Msg
                  v

                P 2
+-----------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  |       |  | intern| |
| |       |  | [Msg] |  |       | |
| |       |  | ^     |  | first | |
```



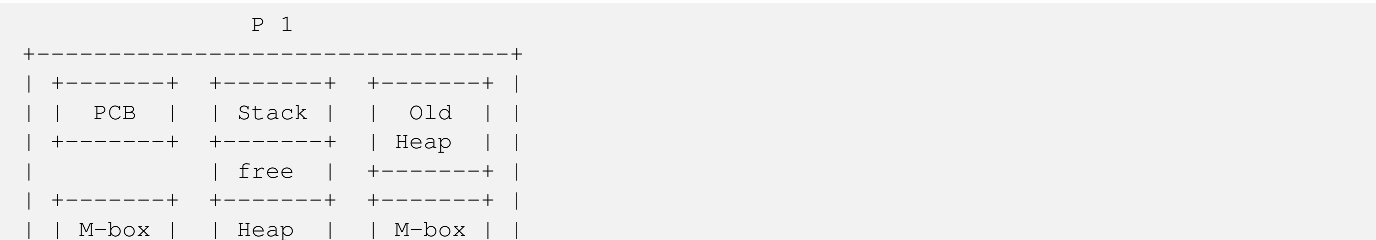
```
| +-------+  +-|-----+  +---|---+ |
|              |            v     |
|              |        +-------+ |
|              |        |next:[]| |
|              |        | m: *  | |
|              |        +----|--+ |
|              |             |    |
|              +-------------+    |
+---------------------------------+
```

If `P1` can not get the `main lock` of P2 or there is not enough space on `P2's` heap and the allocation strategy is *on_heap* the message will end up in an `m-buf` but linked from the internal mailbox:

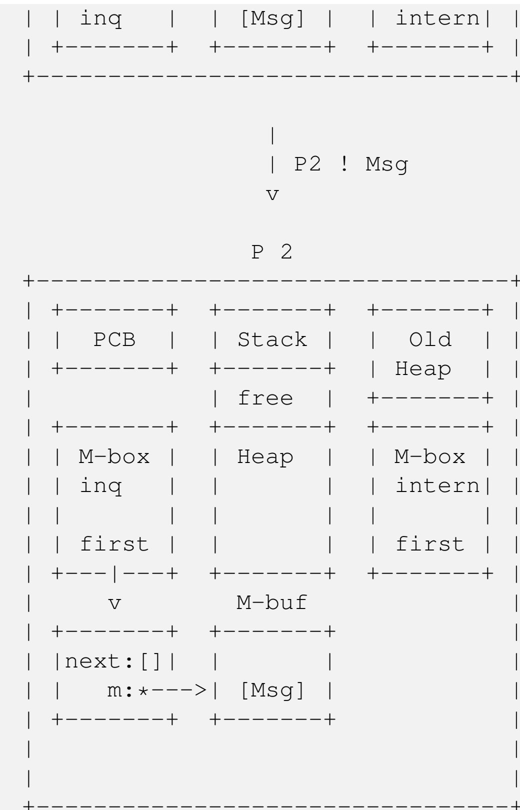
```
                P 1
+---------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  | [Msg] |  | intern| |
| +-------+  +-------+  +-------+ |
+---------------------------------+

                 |
                 | P2 ! Msg
                 v

                P 2
+---------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  |       |  | intern| |
| |       |  |       |  |       | |
| |       |  |       |  | first | |
| +-------+  +-------+  +---|---+ |
|             M-buf         v     |
|            +-------+  +-------+ |
|         +->| [Msg] |  |next:[]| |
|         |  |       |  | m: *  | |
|         |  +-------+  +----|--+ |
|         |                  |    |
|         +------------------+    |
+---------------------------------+
```

After a GC the message will be moved into the heap.

If the allocation strategy is *off_heap* the message will end up in an `m-buf` and linked from the external mailbox:



```
                P 1
+---------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
```



```
| | inq    |  | [Msg] |  | intern| |
| +-------+  +-------+  +-------+ |
+---------------------------------+

                 |
                 | P2 ! Msg
                 v

                P 2
+---------------------------------+
| +-------+  +-------+  +-------+ |
| |  PCB  |  | Stack |  |  Old  | |
| +-------+  +-------+  | Heap  | |
|            | free  |  +-------+ |
| +-------+  +-------+  +-------+ |
| | M-box |  | Heap  |  | M-box | |
| | inq   |  |       |  | intern| |
| |       |  |       |  |       | |
| | first |  |       |  | first | |
| +---|---+  +-------+  +-------+ |
|     v        M-buf              |
| +-------+  +-------+            |
| |next:[]|  |       |            |
| |   m:*--->| [Msg] |            |
| +-------+  +-------+            |
|                                 |
|                                 |
+---------------------------------+
```

After a GC the message will still be in the `M-buf`. Not until the message is received and reachable from some other object on the heap or from the stack will the message be copied to the process heap during a GC.

3.6.4 Receiving a Message

Erlang supports selective receive, which means that a message that doesn't match can be left in the mailbox for a later receive. And the processes can be suspended with messages in the mailbox when no message matches. The `msg.save` field contains a pointer to a pointer to the next message to look at.

In later chapters we will cover the details of `m-bufs` and how the garbage collector handles mailboxes. We will also go through the details of how receive is implemented in the BEAM in later chapters.

3.6.5 Tuning Message Passing

With the new *message_queue_data* flag introduced in Erlang 19 you can trade memory for execution time in a new way. If the receiving process is overloaded and holding on to the `main lock`, it might be a good strategy to use the *off_heap* allocation in order to let the sending process quickly dump the message in an `M-buf`.

If two processes have a nicely balanced producer consumer behavior where there is no real contention for the process lock then allocation directly on the receivers heap will be faster and use less memory.

If the receiver is backed up and is receiving more messages than it has time to handle, it might actually start using more memory as messages are copied to the heap, and migrated to the old heap. Since unseen messages are considered live, the heap will need to grow and use more memory.

In order to find out which allocation strategy is best for your system you will need to benchmark and measure the behavior. The first and easiest test to do is probably to change the default allocation strategy at the start of the system. The ERTS flag *hmqd* sets the default strategy to either *off_heap* or *on_heap*. If you start Erlang without this flag the default will be *on_heap*. By setting up your benchmark so that Erlang is started with *+hmqd off_heap* you can test whether the system behaves better or worse if all processes use off heap allocation. Then you might want to find bottle neck processes and test switching allocation strategies for those processes only.

3.7 The Process Dictionary

There is actually one more memory area in a process where Erlang terms can be stored, the *Process Dictionary*.

The *Process Dictionary* (PD) is a process local key-value store. One advantage with this is that all keys and values are stored on the heap and there is no copying as with send or an ETS table.

We can now update our view of a process with yet another memory area, PD, the process dictionary:

```
+-------+             +-------+  +-------+
|  PCB  |             | Stack |  |  PD   |
+-------+             +-------+  +-------+
                      | free  |
+-------+  +-------+  +-------+  +-------+
| M-box |  | M-box |  | Heap  |  |  Old  |
| intern|  | inq   |  |       |  | Heap  |
+-------+  +-------+  +-------+  +-------+

+-------+  +-------+  +-------+  +-------+
| m-buf |  | m-buf |  | m-buf |  | m-buf |
+-------+  +-------+  +-------+  +-------+
```

With such a small array you are bound to get some collisions before the area grows. Each hash value points to a bucket with key value pairs. The bucket is actually an Erlang list on the heap. Each entry in the list is a two tuple (*{key, Value}*) also stored on the heap.

Putting an element in the PD is not completely free, it will result in an extra tuple and a cons, and might cause garbage collection to be triggered. Updating a key in the dictionary, which is in a bucket, causes the whole bucket (the whole list) to be reallocated to make sure we don't get pointers from the old heap to the new heap. (In <ref linkend="ch.memory"/> we will see the details of how garbage collection works.)

3.8 Dig In

In this chapter we have looked at how a process in implemented. In particular we looked at how the memory of a process is organized, how message passing works and the information in the PCB. We also looked at a number of tools for inspecting processes introspection, such as *erlang:process_info*, and the *hipe:show**_ bifs.

Use the functions `erlang:processes/0` and `erlang:process_info/1/2` to inspect the processes in the system. Here are some functions to try:

```
1> Ps = erlang:processes().
[<0.0.0>,<0.3.0>,<0.6.0>,<0.7.0>,<0.9.0>,<0.10.0>,<0.11.0>,
 <0.12.0>,<0.13.0>,<0.14.0>,<0.15.0>,<0.16.0>,<0.17.0>,
 <0.19.0>,<0.20.0>,<0.21.0>,<0.22.0>,<0.23.0>,<0.24.0>,
 <0.25.0>,<0.26.0>,<0.27.0>,<0.28.0>,<0.29.0>,<0.33.0>]
2> P = self().
<0.33.0>
3> erlang:process_info(P).
[{current_function,{erl_eval,do_apply,6}},
 {initial_call,{erlang,apply,2}},
 {status,running},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.27.0>]},
 {dictionary,[]},
 {trap_exit,false},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.26.0>},
 {total_heap_size,17730},
```

```
 {heap_size,6772},
 {stack_size,24},
 {reductions,25944},
 {garbage_collection,[{min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,1}]},
 {suspending,[]}]
 4>  lists:keysort(2,[{P,element(2,erlang:process_info(P,
     total_heap_size))} || P <- Ps]).
[{<0.10.0>,233},
 {<0.13.0>,233},
 {<0.14.0>,233},
 {<0.15.0>,233},
 {<0.16.0>,233},
 {<0.17.0>,233},
 {<0.19.0>,233},
 {<0.20.0>,233},
 {<0.21.0>,233},
 {<0.22.0>,233},
 {<0.23.0>,233},
 {<0.25.0>,233},
 {<0.28.0>,233},
 {<0.29.0>,233},
 {<0.6.0>,752},
 {<0.9.0>,752},
 {<0.11.0>,1363},
 {<0.7.0>,1597},
 {<0.0.0>,1974},
 {<0.24.0>,2585},
 {<0.26.0>,6771},
 {<0.12.0>,13544},
 {<0.33.0>,13544},
 {<0.3.0>,15143},
 {<0.27.0>,32875}]
9>
```

Chapter 4

The Erlang Type System and Tags

One of the most important aspects of ERTS to understand is how ERTS stores data, that is, how Erlang terms are stored in memory. This gives you the basis for understanding how garbage collection works, how message passing works, and gives you an insight into how much memory is needed.

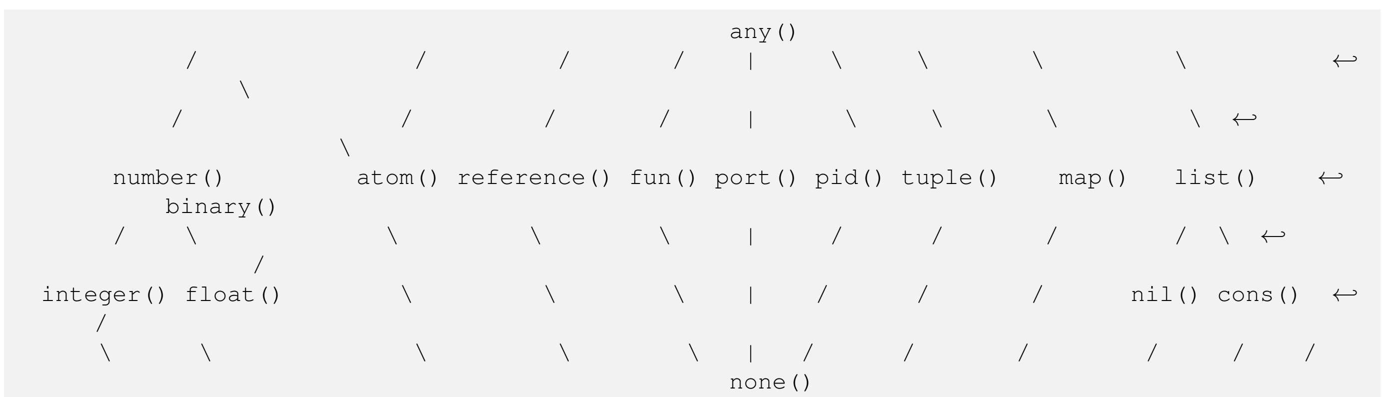
In this chapter you will learn the basic data types of Erlang and how they are implemented in ERTS. This knowledge will be essential in understanding the chapter on memory allocation and garbage collection, see Chapter 12.

4.1 The Erlang Type System

Erlang is *strongly typed*. That is, there is no way to coerce one type into another type, you can only convert from one type to another. Compare this to e.g. C where you can coerce a *char* to an *int* or any type pointed to by a pointer to (*void **).

The Erlang type lattice is quite flat, there are only a few real sub types, numbers have the sub types integer and float, and list has the subtypes nil and cons. (One could also argue that tuple has one subtype for each size.)

The Erlang Type Lattice



```
                                           any()
         /          /      /     /    |    \    \     \      \        ↩
           \
        /          /      /     /     |     \    \     \      \  ↩
                    \
     number()        atom() reference() fun() port() pid() tuple()   map()   list()    ↩
       binary()
    /   \         \      \     \      |    /     /      /       /  \  ↩
              /
 integer() float()     \      \      \    |   /     /      /      nil() cons()  ↩
   /
   \    \          \      \      \   |  /     /      /      /    /   /
                                           none()
```

There is a partial order (< and >) on all terms in Erlang where the types are ordered from left to right in the above lattice.

The order is partial and not total since integers and floats are converted before comparison. Both (1 < 1.0) and (1.0 < 1) are false, and (1 =< 1.0 and 1 >= 1.0) and (1 =/= 1.0). The number with the lesser precision is converted to the number with higher precision. Usually integers are converted to floats. For very large or small floats the float is converted to an integer. This happens if all significant digits are to the left of the decimal point.

Since Erlang 18, when two maps are compared for order they are compared as follows: If one map has fewer elements than the other it is considered smaller. Otherwise the keys are compared in term order, where all integers are considered smaller than all floats. If all the keys are the same then each value pair (in key order) is compared arithmetically, i.e. by first converting them to the same precision.

The same is true when comparing for equality, thus #{1 ⇒ 1.0} == #{1 ⇒ 1} but #{1.0 ⇒ 1} /= #{1 ⇒ 1}.

In Erlang versions prior to 18 keys where also compared arithmetically.

Erlang is dynamically typed. That is, types will be checked at runtime and if a type error occurs an exception is thrown. The compiler does not check the types at compile time, unlike in a statically typed language like C or Java where you can get a type error during compilation.

These aspects of the Erlang type system, strongly statically typed with an order on the types puts some constraints on the implementation of the language. In order to be able to check and compare types at runtime each Erlang term has to carry its type with it.

This is solved by *tagging* the terms.

4.2 The Tagging Scheme

In the memory representation of an Erlang term a few bits are reserved for a type tag. For performance reasons the terms are divided into *immediates* and *boxed* terms. An immediate term can fit into a machine word, that is, in a register or on a stack slot. A boxed term consists of two parts: a tagged pointer and a number of words stored on the process heap. The *boxes* stored on the heap have a header and a body, unless it is a list.

Currently ERTS uses a staged tag scheme, the history and reasoning behind the this scheme is explained in a technical report from the HiPE group. (See link:http://www.it.uu.se/research/publications/reports/2000-029/) The tagging scheme is implemented in `erl_term.h`.

The basic idea is to use the least significant bits for tags. Since most modern CPU architectures aligns 32- and 64-bit words, there are at least two bits that are "unused" for pointers. These bits can be used as tags instead. Unfortunately those two bits are not enough for all the types in Erlang, more bits are therefore used as needed.

4.2.1 Tags for Immediates

The first two bits (the primary tag) are used as follows:

```
00 Header (on heap) CP (on stack)
01 List (cons)
10 Boxed
11 Immediate
```

The header tag is only used on the heap for header words, more on that later. On the stack 00 indicates a return address. The list tag is used for cons cells, and the boxed tag is used for all other pointers to the heap. The immediate tag is used further divided like this:

```
00 11 Pid
01 11 Port
10 11 Immediate 2
11 11 Small integer
```

Pid and ports are immediates and can be compared for equality efficiently. They are of course in reality just references, a pid is a process identifier and it points to a process. The process does not reside on the heap of any process but is handled by the PCB. A port works in much the same way.

There are two types of integers in ERTS, small integers and bignums. Small integers fits in one machine word minus four tag bits, i.e. in 28 or 60 bits for 32 and 64 bits system respectively. Bignums on the other hand can be as large as needed (only limited by the heap space) and are stored on the heap, as boxed objects.

By having all four tag bits as ones for small integers the emulator can make an efficient test when doing integer arithmetic to see if both arguments are immediates. (`is_both_small(x,y)` is defined as `(x & y & 1111) ==1111`).

The Immediate 2 tag is further divided like this:

```
00 10 11 Atom
01 10 11 Catch
10 10 11   [UNUSED]
11 10 11 Nil
```

Atoms are made up of an index in the *atom table* and the atom tag. Two atom immediates can be compared for equality by just comparing their immediate representation.

In the atom table atoms are stored as C structs like this:

```
typedef struct atom {
    IndexSlot slot;  /* MUST BE LOCATED AT TOP OF STRUCT!!! */
    int len;         /* length of atom name */
    int ord0;        /* ordinal value of first 3 bytes + 7 bits */
    byte* name;      /* name of atom */
} Atom;
```

Thanks to the `len` and the `ord0` fields the order of two atoms can be compared efficiently as long as they don't start with the same four letters.

Note

If you for some reason generate atoms with a pattern like name followed by a number and then store them in an ordered list or ordered tree the atom comparison will be more expensive if they all have the same first letters (e.g. foo_1, foo_2, etc.).

Not that you should ever generate atom names, since the atom table is limited. I'm just saying, there is an evil micro optimization to be found here.

You would of course never do this, but if you find code that generates atom with a number followed by a postfix name, now you know what the author of that code might have been thinking.

The Catch immediate is only used on the stack. It contains an indirect pointer to the continuation point in the code where execution should continue after an exception. More on this in Chapter 8.

The Nil tag is used for the empty list (nil or `[]`). The rest of the word is filled with ones.

4.2.2 Tags for Boxed Terms

Erlang terms stored on the heap uses several machine words. Lists, or cons cells, are just two consecutive words on the heap. The head and the tail (or car and cdr as they are called in lisp and some places in the ERTS code).

A string in Erlang is just a list of integers representing characters. In releases prior to Erlang OTP R14 strings have been encoded as ISO-latin-1 (ISO8859-1). Since R14 strings are encoded as lists of Unicode code points. For strings in latin-1 there is no difference since latin-1 is a subset of Unicode.

The string "Hello" might look like this in memory:

Representation of the string "Hello" on a 32 bit machine.

```
hend ->     +-------- -------- -------- --------+
            |              ...                  |
            |              ...                  |
            |00000000 00000000 00000000 10000001| 128 + list tag  ---------------+
stop ->     |                                   |                                |
                                                                                 |
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 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 ->     +-----------------------------------+
```

All other boxed terms start with a header word. The header word uses a four bit header tag and the primary header tag (00), it also has an arity which says how many words the boxed term uses. On a 32-bit machine it looks like this: `aaaaaaaaaaaaaaaaaaaaaaaaaatttt00`.

The tags are:

```
0000   ARITYVAL (Tuples)
0001   BINARY_AGGREGATE                |
001s   BIGNUM with sign bit            |
0100   REF                             |
0101   FUN                             | THINGS
0110   FLONUM                          |
0111   EXPORT                          |
1000   REFC_BINARY     |               |
1001   HEAP_BINARY     | BINARIES      |
1010   SUB_BINARY      |               |
1011     [UNUSED]
1100   EXTERNAL_PID  |                 |
1101   EXTERNAL_PORT | EXTERNAL THINGS |
1110   EXTERNAL_REF  |                 |
1111   MAP
```

Tuples are stored on the heap with just the arity and then each element in the following words. The empty tuple `{}` is stored just as the word 0 (header tag 0, tuple tag 0000, and arity 0).

Representation of the tuple {104,101,108,108,111} on a 32 bit machine.

```
hend ->     +-------- -------- -------- --------+
            |              ...                  |
            |              ...                  |
            |00000000 00000000 00000000 10000010| 128 + boxed tag ---------------+
stop ->     |                                   |                                |
                                                                                 |
htop ->     |                                   |                                |
        150 |00000000 00000000 00000110 11111111| (o) 111 bsl 4 + small int tag  |
        144 |00000000 00000000 00000110 11001111| (l) 108 bsl 4 + small int tag  |
        140 |00000000 00000000 00000110 11001111| (l) 108 bsl 4 + small int tag  |
        136 |00000000 00000000 00000110 01011111| (e) 101 bsl 4 + small int tag  |
        132 |00000000 00000000 00000110 10001111| (H) 104 bsl 4 + small int tag  |
        128 |00000000 00000000 00000000 10100000| 5 bsl 6 + tuple & header tag <-+
            |                ...                |
heap ->     +-----------------------------------+
```

A *binary* is an imutable array of bytes. There are four types of internal representations of *binaries*. The two types *heap binaries* and *refc binaries* contains binary data. The other two types, *sub binaries* and *match contexts* (the BINARY_AGREGATE tag) are smaller references into one of the other two types.

Binaries that are 64 bytes or less can be stored directly on the process heap as *heap binaries*. Larger binaries are reference counted and the paylod is stored outside of the process heap, a reference to the payload is stoed on the process heap in an object called a *ProcBin*.

We will talk more about binaries in the Chapter 12.

Integers that do not fit in a small integer (word size - 4 bits) are stored on the heap as "bignums" (or arbitrary precision integers). A bignum has a header word followed by a number of words encoding the bignum. The sign part of the bignum tag (s) in the header encodes the sign of the number (s=0 for positive numbers, and s=1 for negative numbers).

TODO: Describe bignum encoding. (And arithmetic ?)

A reference is a *"unique"* term often used to tag messages in order to basically implement a channel over a process mailbox. A references is implemented as an 82 bit counter. After 9671406556917033397649407 calls to `make_ref` the counter will wrap and start over with ref 0 again. You need a really fast machine to do that many calls to `make_ref` within your lifetime. Unless you restart the node, in which case it also will start from 0 again, but then all the old local refs are gone. If you send the pid to another node it becomes an external ref, see below.

On a 32-bit system a local ref takes up four 32-bit words on the heap. On a 64-bit system a ref takes up three 64-bit words on the heap.

Representation of a ref in a 32-bit (or half-word) system.

```
    |00000000 00000000 00000000 11010000| Arity 3 + ref tag
    |00000000 000000rr rrrrrrrr rrrrrrrr| Data0
    |rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr| Data1
    |rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr| Data2
```

The reference number is (Data2 bsl 50) + (Data1 bsl 18) + Data0.

Outline

TODO

```
The implementation of floats,  ports, pids. Strings as lists, IO lists,
lists on 64-bit machines. Binaries, sub binaries, and copying. Records.

Possibly: The half-word machine. Sharing and deep copy. (or this will be in GC ↩
    )

Outro/conclusion
```

Chapter 5

The Erlang Virtual Machine: BEAM

BEAM (Bogumil's/Björn's Abstract Machine) is the machine that executes the code in the Erlang Runtime System. It is a garbage collecting, reduction counting, virtual, non-preemptive, directly threaded, register machine. If that doesn't tell you much, don't worry, in the following sections we will go through what each of those words means in this context.

The virtual machine, BEAM, is at the heart of the Erlang node. It is the BEAM that executes the Erlang code. That is, it is BEAM that executes your application code. Understanding how BEAM executes the code is vital to be able to profile and tune your code.

The BEAM desig influences large parts of the rest of ERTS. The primitives for scheduling infuences the Scheduler (Chapter 11), the representation of Erlang terms and the interaction with the memory infuences the Garbage Collector (Chapter 12). By understanding the basic design of BEAM you will more easily understand the implementations of these other components.

5.1 Working Memory: A stack machine, it is not

As opposed to its predecessor Jam (Joe's Abstract Machine) which was a stack machine, the BEAM is a register machine loosely based on WAM (TODO: cite). In a stack machine each operand to an instruction is first pushed to the working stack, then the instruction pops its arguments and then it pushes the result on the stack.

Stack machines are quite popular among virtual machine- and programming language implementers since they are quite easy to generate code for, and the code becomes very compact. The compiler does not need to do any register allocation, and most operations do not need any arguments (in the instruction stream).

Compiling the expression "8 + 17 * 2." to a stack machine could yield code like:

```
push 8
push 17
push 2
multiply
add
```

This code can be generated directly from the parse tree of the expression. By using Erlang expression and the modules `erl_scan` and `erl_parse` we can build the world's most simplistic compiler.

```
compile(String) ->
    [ParseTree] = element(2,
                          erl_parse:parse_exprs(
                            element(2,
                                    erl_scan:string(String)))),
    generate_code(ParseTree).

generate_code({op, _Line, '+', Arg1, Arg2}) ->
    generate_code(Arg1) ++ generate_code(Arg2) ++ [add];
generate_code({op, _Line, '*', Arg1, Arg2}) ->
```

```
    generate_code(Arg1) ++ generate_code(Arg2) ++ [multiply];
generate_code({integer, _Line, I}) -> [push, I].
```

And an even more simplistic virtual stack machine:

```
interpret(Code) -> interpret(Code, []).

interpret([push, I |Rest], Stack)              -> interpret(Rest, [I|Stack]);
interpret([add     |Rest], [Arg2, Arg1|Stack]) -> interpret(Rest, [Arg1+Arg2|Stack]);
interpret([multiply|Rest], [Arg2, Arg1|Stack]) -> interpret(Rest, [Arg1*Arg2|Stack]);
interpret([],              [Res|_])            -> Res.
```

And a quick test run gives us the answer:

```
1> stack_machine:interpret(stack_machine:compile("8 + 17 * 2.")).
42
```

Great, you have built your first virtual machine! Handling subtraction, division and the rest of the Erlang language is left as an exercise for the reader.

Anyway, the BEAM is **not** a stack machine, it is a *register machine*. In a register machine instruction operands are stored in registers instead of the stack, and the result of an operation usually end up in a specific register.

Most register machines do still have a stack used for passing arguments to functions and saving return addresses. BEAM has both a stack and registers, but just as in WAM the stack slots are accessible through registers called Y-registers. BEAM also have a number of X-registers, and a special register X0 (sometimes also called R0) which works as an accumulator where results are stored.

The X registers are used as argument registers for function calls and register X0 is used for the return value.

The X registers are stored in a C-array in the BEAM emulator and they are globally accessible from all functions. The X0 register is cashed in a local variable mapped to a physical machine register in the native machine on most architectures.

The Y registers are stored in the stack frame of the caller and only accessible by the calling functions. To save a value across a function call BEAM allocates a stack slot for it in the current stack frame and then moves the value to a Y register.

```
  hend ->  +----+    -
           |....|
  (fp) ->  | AN |
  (y0) ->  |    |
  (y1) ->  |    |
  stop ->  |    |
           |    |
           |    |
  htop ->  |    |
           |....|
           |....|
  heap ->  +----+

           +----+
   X1000   |    |
    X999   |    |
   ...      ....
      X2   |    |
      X1   |    |
     (X0)  |    |
           +----+
```

Let us compile the following program with the *'S'* flag:

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

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

id(I) -> I.
```

Then we get the following code for the add function:

```
{function, add, 2, 2}.
  {label,1}.
    {func_info,{atom,add},{atom,add},2}.
  {label,2}.
    {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.
```

Here we can see that the code (starting at label 2) first allocates a stack slot, to get space to save the argument *B* over the function call *id(A)*. The value is then saved by the instruction *{move,{x,1},{y,0}}* (read as move x1 to y0 or in imperative style: y0 := x1).

The id function (at label f4) is then called by *{call,1,{f,4}}*. (We will come back to what the argument "1" stands for later.) Then the result of the call (now in X0) need to be saved on the stack (Y0), but the argument *B* is saved in Y0 so the BEAM does a bit of shuffling:

Except for the x and y registers, there are a number of special purpose registers:

SPECIAL PURPOSE REGISTERS

- Htop - The top of the heap.
- E - The top of the stack.
- CP - Continuation Pointer, i.e. function return address
- I - instruction pointer
- fcalls - reduction counter

These registers are cached versions of the corresponding fields in the PCB.

```
    {move,{x,0},{x,1}}. % x1 := x0 (id(A))
    {move,{y,0},{x,0}}. % x0 := y0 (B)
    {move,{x,1},{y,0}}. % y0 := x1 (id(A))
```

Now we have the second argument *B* in x0 (the first argument register) and we can call the *id* function again *{call,1,{f,4}}*.

After the call x0 contains *id(B)* and y0 contains *id(A)*, now we can do the addition: *{gc_bif,+,{f,0},1,[{y,0},{x,0}],{x,0}}*. (We will go into the details of bif calls and gc later.)

5.2 Dispatch: Directly Threaded Code

The instruction decoder in BEAM is implemented with a technique called *directly threaded* code. In this context the word *threaded* has nothing to do with OS threads, concurrency or parallelism. It is the execution path which is threaded through the virtual machine itself.

If we take a look at our naive stack machine for arithmetic expressions we see that we use Erlang atoms and pattern matching to decode which instruction to execute. This is a very heavy machinery to just decode machine instructions. In a real machine we would code each instruction as a "machine word" integer.

We can rewrite our stack machine to be a *byte code* machine implemented in C. First we rewrite the compiler so that it produces byte codes. This is pretty straight forward, just replace each instruction encoded as an atom with a byte representing the instruction. To be able to handle integers larger than 255 we encode integers with a size byte followed by the integer encoded in bytes.

```
compile(Expression, FileName) ->
    [ParseTree] = element(2,
                          erl_parse:parse_exprs(
                            element(2,
                                    erl_scan:string(Expression)))),
    file:write_file(FileName, generate_code(ParseTree) ++ [stop()]).

generate_code({op, _Line, '+', Arg1, Arg2}) ->
    generate_code(Arg1) ++ generate_code(Arg2) ++ [add()];
generate_code({op, _Line, '*', Arg1, Arg2}) ->
    generate_code(Arg1) ++ generate_code(Arg2) ++ [multiply()];
generate_code({integer, _Line, I}) -> [push(), integer(I)].

stop()     -> 0.
add()      -> 1.
multiply() -> 2.
push()     -> 3.
integer(I) ->
    L = binary_to_list(binary:encode_unsigned(I)),
    [length(L) | L].
```

Now lets write a simple virtual machine in C. The full code can be found in [appendix].

```
#define STOP 0
#define ADD  1
#define MUL  2
#define PUSH 3

#define pop()   (stack[--sp])
#define push(X) (stack[sp++] = X)

int run(char *code) {
  int stack[1000];
  int sp = 0, size = 0, val = 0;
  char *ip = code;

  while (*ip != STOP) {
    switch (*ip++) {
    case ADD: push(pop() + pop()); break;
    case MUL: push(pop() * pop()); break;
    case PUSH:
      size = *ip++;
      val = 0;
      while (size--) { val = val * 256 + *ip++; }
      push(val);
      break;
    }
  }
  return pop();
}
```

You see, a virtual machine written in C does not need to be very complicated. This machine is just a loop checking the byte code at each instruction by looking at the value pointed to by the _instruction pointer _ (ip).

For each byte code instruction it will switch on the instruction byte code and jump to the case which executes the instruction. This requires a decoding of the instruction and then a jump to the correct code. If we look at the assembly for vsm.c (`gcc -S vsm.c`) we see the inner loop of the decoder:

```
L11:
        movl    -16(%ebp), %eax
        movzbl  (%eax), %eax
        movsbl  %al, %eax
        addl    $1, -16(%ebp)
        cmpl    $2, %eax
        je      L7
        cmpl    $3, %eax
        je      L8
        cmpl    $1, %eax
        jne     L5
```

It has to compare the byte code with each instruction code and then do a conditional jump. In a real machine with many instructions this can become quite expensive.

A better solution would be to have a table with the address of the code then we could just use an index into the table to load the address and jump without the need to do a compare. This technique is sometimes called *token threaded code*. Taking this a step further we can actually store the address of the function implementing the instruction in the code memory. This is called *subroutine threaded code*.

This approach will make the decoding simpler at runtime, but it makes the whole VM more complicated by requiring a loader. The loader replaces the byte code instructions with addresses to functions implementing the instructions.

A loader might look like:

```
typedef void (*instructionp_t)(void);

instructionp_t *read_file(char *name) {
  FILE *file;
  instructionp_t *code;
  instructionp_t *cp;
  long  size;
  char ch;
  unsigned int val;

  file = fopen(name, "r");

  if(file == NULL) exit(1);

  fseek(file, 0L, SEEK_END);
  size = ftell(file);
  code = calloc(size, sizeof(instructionp_t));
  if(code == NULL) exit(1);
  cp = code;

  fseek(file, 0L, SEEK_SET);
  while ( ( ch = fgetc(file) ) != EOF )
    {
      switch (ch) {
      case ADD: *cp++ = &add; break;
      case MUL: *cp++ = &mul; break;
      case PUSH:
        *cp++ = &pushi;
        ch = fgetc(file);
        val = 0;
        while (ch--) { val = val * 256 + fgetc(file); }
        *cp++ = (instructionp_t) val;
        break;
      }
    }
  *cp = &stop;
```

```
  fclose(file);
  return code;
}
```

As we can see, we do more work at load time here, including the decoding of integers larger than 255. (Yes, I know, the code is not safe for very large integers.)

The decode and dispatch loop of the VM becomes quite simple though:

```
int run() {
  sp = 0;
  running = 1;

  while (running) (*ip++)();

  return pop();
}
```

Then we just need to implement the instructions:

```
void add()  { int x,y; x = pop(); y = pop(); push(x + y); }
void mul()  { int x,y; x = pop(); y = pop(); push(x * y); }
void pushi(){ int x;   x = (int)*ip++;       push(x); }
void stop() { running = 0; }
```

In BEAM this concept is taken one step further, and BEAM uses *directly threaded code* (sometimes called only *thread code*). In directly threaded code the call and return sequence is replaced by direct jumps to the implementation of the next instruction. In order to implement this in C, BEAM uses the GCC extension "labels as values".

We will look closer at the BEAM emulator later but we will take a quick look at how the add instruction is implemented. The code is somewhat hard to follow due to the heavy usage of macros. The STORE_ARITH_RESULT macro actually hides the dispatch function which looks something like: *I += 4; Goto(*I);*.

```
#define OpCase(OpCode)    lb_##OpCode
#define Goto(Rel) goto *(Rel)

...

 OpCase(i_plus_jId):
 {
     Eterm result;

     if (is_both_small(tmp_arg1, tmp_arg2)) {
         Sint i = signed_val(tmp_arg1) + signed_val(tmp_arg2);
         ASSERT(MY_IS_SSMALL(i) == IS_SSMALL(i));
         if (MY_IS_SSMALL(i)) {
             result = make_small(i);
             STORE_ARITH_RESULT(result);
         }

     }
     arith_func = ARITH_FUNC(mixed_plus);
     goto do_big_arith2;
 }
```

To make it a little easier to understand how the BEAM dispatcher is implemented let us take a somewhat imagenary example. We will start with some real external BEAM code but then I will invent some internal BEAM instructions and implement them in C.

If we start with a simple add function in Erlang:

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

Compiled to BEAM code this will look like:

```
{function, add, 2, 2}.
  {label,1}.
    {func_info,{atom,add},{atom,add},2}.
  {label,2}.
    {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.
```

(See add.erl and add.S in Appendix .14 for the full code.)

Now if we zoom in on the the three instructions betwen the function calls in this code:

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

This code first saves the return value of the function call (x0) in a new register (x1). Then it moves the caller saves register (y0) to the first argument register (x0). Finally it moves the saved value in x1 to the caller save register (y0) so that it will survive the next function call.

Imagine that we would implement three instruction in BEAM called `move_xx`, `move_yx`, and `move_xy` (These instructions does not exist in the BEAM we just use them to illustrate this example):

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


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

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


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

Note that the star in "`goto *`" does not mean dereference, the expression means jump to an address pointer, we should really write it as "`goto*`".

Now imagine that the compile C code for these instructions end up at memory addresses 0x3000, 0x3100, and 0x3200. When the BEAM code is loaded the three move instructions in the code will be replaced by the memory addresses of the implementation of the instructions. Imagine that the code (`{move,{x,0},{x,1}}, {move,{y,0},{x,0}}, {move,{x,1},{y,0}}`) is loaded at address 0x1000:

```
                       /  0x1000: 0x3000 -> 0x3000: OpCase(move_xx): x(Arg(1)) = x(Arg(0))
{move,{x,0},{x,1}}  {     0x1004: 0x0                                I += 3;
                       \  0x1008: 0x1                                Goto(*I);
                       /  0x100c: 0x3200
{move,{y,0},{x,0}}  {     0x1010: 0x1
                       \  0x1014: 0x1
                       /  0x1018: 0x3100
{move,{y,0},{x,0}}  {     0x101c: 0x1
                       \  0x1020: 0x1
```

The word at address 0x1000 points to the implementation of the move_xx instruction. If the register `I` contains the instruction pointer, pointing to 0x1000 then the dispatch will be to fetch `*I` (i.e. 0x3000) and jump to that address. ("`goto* *I`")

In Chapter 7 we will look more closely at some real BEAM instructions and how they are implemented.

5.3 Scheduling: Non-preemptive, Reduction counting

Most modern multi-threading operating systems use preemptive scheduling. This means that the operating system decides when to switch from one process to another, regardless of what the process is doing. This protects the other processes from a processes misbehaving by not yielding in time.

In cooperative multitasking which uses a non-preemptive scheduler the running process decides when to yield. This has the advantage that the yielding process can do so in a known state.

For example in a language such as Erlang with dynamic memory management and tagged values, an implementation may be designed such that a process only yields when there are no untagged values in working memory.

Take the add instruction as an example, to add two Erlang integers, the emulator first has to untag the integers, then add them together and then tag the result as an integer. If a fully preemptive scheduler is used there would be no guarantee that the process isn't suspended while the integers are untagged. Or the process could be suspended while it is creating a tuple on the heap, leaving us with half a tuple. This would make it very hard to traverse a suspended process stack and heap.

On the language level all processes are running concurrently and the programmer should not have to deal with explicit yields. BEAM solves this by keeping track of how long a process has been running. This is done by counting *reductions*. The term originaly comes from the mathematical term beta-reduction used in lambda calculus.

The definition of a reduction in BEAM not very specific, but we can see it as a small piece of work, which shouldn't take *too long*. Each function call is counted as a reduction, and BEAM does a test upon entry to each function if the process has used up all its reductions.

Since there are no loops in Erlang, only tail-recursive function calls, it is very hard to write a program that does any significant amount of work without using up its reductions.

Warning

There are some bifs that can run for a long time only using 1 reduction, like `term_to_binary` and `binary_to_term`. Try to make sure that you only call these biffs with small terms or binaries, or you might lock up the scheduler for a very long time.

Also, if you write your own Nifs, make sure they can yield and that they bump the reduction counter by an amount proportional to their run time.

We will go through the details of how the scheduler works in Chapter 11.

5.4 Memory Management: Garbage Collecting

Erlang supports garbage collection; as an Erlang progammer you do not need to do explicit memory management. On the BEAM level, though, the code is responsible for checking for stack and heap overrun, and for allocating enough space on the stack and the heap.

The BEAM instruction `test_heap` will ensure that there is as much space on the heap as requested. If needed the instruction will call the garbage collector to reclaim space on the heap. The garbage collector in turn will call the lower levels of the memory subsystem to allocate or free memory as needed. We will look at the details of memory management and garbage collection in Chapter 12.

5.5 BEAM: it is virtually unreal

The BEAM is a virtual machine, by that we mean that it is implemented in software instead of in hardware. There has been projects to implement the BEAM by FPGA, and there is nothing stopping anyone from implementing the BEAM in hardware. A better description might be to call the BEAM an Abstract machine, and see it as blueprint for a machine which can execute BEAM code. And, in fact, the "am" in BEAM stands for "Abstract Machine".

In this book we will make no distinction between abstract machines, and virtual machines or their implementation. In a more formal setting an abstract machine is a theoretical model of a computer, and a virtual machine is either a software implementation of an abstract machine or a software emulator of a real physical machine.

Unfortunately there exist no official specification of the BEAM, it is currently only defined by the implementation in Erlang/OTP. If you want to implement your own BEAM you would have to try to mimic the current implementation not knowing which parts are essential and which parts are accidental. You would have to mimic every observable behavior to be sure that you have a valid BEAM interpreter.

TODO: Conclusion and handover to the chapters on instructions.

Chapter 6

Modules and The BEAM File Format (16p)

6.1 Modules

What is a module. How is code loaded. How does hot code loading work. How does the purging work. How does the code server work. How does dynamic code loading work, the code search path. Handling code in a distributed system. (Overlap with chapter 10, have to see what goes where.) Parameterized modules. How p-mods are implemented. The trick with p-mod calls. Here follows an excerpt from the current draft:

6.2 The BEAM File Format

The definite source of information about the beam file format is obviously the source code of beam_lib.erl (see https://github.com/-erlang/otp/blob/maint/lib/stdlib/src/beam_lib.erl). There is actually also a more readable but slightly dated description of the format written by the main developer and maintainer of Beam (see http://www.erlang.se/~bjorn/beam_file_format.html).

The beam file format is based on the interchange file format (EA IFF)#, with two small changes. We will get to those shortly. An IFF file starts with a header followed by a number of “chunks”. There are a number of standard chunk types in the IFF specification dealing mainly with images and music. But the IFF standard also lets you specify your own named chunks, and this is what BEAM does.

An IFF file looks like this:

```
{'FORM':4
 SIZE:4
 FORMTYPE:4
 [{CHUNKNAME:4
   CHUNKSIZE:4
   [CHUNKDATA:1]:CHUNKSIZE
   [2-BYTEPAD:1]:0..1
  }
 ]:SIZE-4
}
```

Beam files differ from standard IFF files, in that each chunk is aligned on 4-byte boundary (i.e. 32 bit word) instead of on a 2-byte boundary as in the IFF standard. To indicate that this isn’t a standard IFF file the IFF header is tagged with “FOR1” instead of “FORM”. The IFF specification suggest this tag for future extensions.

The form type used by beam is “BEAM”. A beam file has the following layout:

```
{'FOR1':4
 SIZE:4
 'BEAM':4
 [{CHUNKNAME:4
```

```
  CHUNKSIZE:4
  [CHUNKDATA:1]:CHUNKSIZE
  [4-BYTEPAD:1]:0..3
  }
 ]:SIZE-4
}
```

This file format prepends all areas with the size of the following area making it easy to parse the file directly while reading it from disk. To illustrate the structure and content of beam files, we will write a small program that extract all the chunks from a beam file. To make this program as simple and readable as possible we will not parse the file while reading, instead we load the whole file in memory as a binary, and then parse each chunk. The first step is to get a list of all chunks:

```
-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)).
```

A sample run might look like:

```
> beamfile:read("beamfile.beam").
{848,
[{"Atom",103,
  <<0,0,0,14,4,102,111,114,49,4,114,101,97,100,4,102,105,
    108,101,9,114,101,97,...>>},
 {"Code",341,
  <<0,0,0,16,0,0,0,0,0,0,0,132,0,0,0,14,0,0,0,4,1,16,...>>},
 {"StrT",8,<<"FOR1BEAM">>},
 {"ImpT",88,<<0,0,0,7,0,0,0,3,0,0,0,4,0,0,0,1,0,0,0,7,...>>},
 {"ExpT",40,<<0,0,0,3,0,0,0,13,0,0,0,1,0,0,0,13,0,0,0,...>>},
 {"LocT",16,<<0,0,0,1,0,0,0,6,0,0,0,2,0,0,0,6>>},
 {"Attr",40,
  <<131,108,0,0,0,1,104,2,100,0,3,118,115,110,108,0,0,...>>},
 {"CInf",130,
  <<131,108,0,0,0,4,104,2,100,0,7,111,112,116,105,111,...>>},
 {"Abst",0,<<>>}]}
```

```
Here we can see the chunk names that beam uses.
```

6.2.1 Atom table chunk

The chunk named Atom is mandatory and contains all atoms referred to by the module. The format of the atom chunk is:

```
{"Atom":4
  CHUNKSIZE:4
```

```
  NUMBEROFATOMS:4
  [{LENGTH:1,
    ATOM:LENGTH}]:NUMBEROFATOMS
  [4-BYTEPAD:1]:0..3
}
```

NOTE: There is a further constraint that the module name must be stored as the first atom in the table.

Let us add a decoder for the atom chunk to our Beam file reader:

```
-module(beamfile).
-export([read/1]).

read(Filename) ->
   {ok, File} = file:read_file(Filename),
   <<"FOR1",
     Size:32/integer,
     "BEAM",
     Chunks/binary>> = File,
   {Size, parse_chunks(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).

parse_chunks([{"Atom", _Size,
             <<_Numberofatoms:32/integer, Atoms/binary>>}
            | Rest], Acc) ->
   parse_chunks(Rest,[{atoms,parse_atoms(Atoms)}|Acc]);
parse_chunks([Chunk|Rest], Acc) -> %% Not yet implemented chunk
   parse_chunks(Rest, [Chunk|Acc]);
parse_chunks([],Acc) -> Acc.

parse_atoms(<<Atomlength, Atom:Atomlength/binary, Rest/binary>>) when Atomlength > 0->
   [list_to_atom(binary_to_list(Atom)) | parse_atoms(Rest)];
parse_atoms(_Alignment) -> [].

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

6.2.2 Export table chunk

The chunk named “ExpT” (for EXPort Table) is mandatory and contains information about which functions are exported.

The format of the chunk is:

```
{"ExpT":4
  CHUNKSIZE:4
  NUMBEROFENTRIES:4
  [{FUNCTION:4,
    ARITY:4,
    LABEL:4
   }]:NUMBEROFENTRIES
}
```

We can extend our parse_chunk function by adding the following clause after the atom handling clause:

```
parse_chunks([{"ExpT", _Size,
             <<_Numberofentries:32/integer, Exports/binary>>}
```

```
             | Rest], Acc) ->
   parse_chunks(Rest,[{exports,parse_exports(Exports)}|Acc]);

...

parse_exports(<<Function:32/integer,
               Arity:32/integer,
               Label:32/integer,
               Rest/binary>>) ->
   [{Function, Arity, Label} | parse_exports(Rest)];
parse_exports(<<>>) -> [].
```

6.2.3 Import table chunk

The chunk named “ImpT” (for IMPort Table) is mandatory and contains information about which functions are imported.

The format of the chunk is:

```
{“ImpT”:4
  CHUNKSIZE:4
  NUMBEROFENTRIES:4
  [{FUNCTION:4,
    ARITY:4,
    LABEL:4
   }]:NUMBEROFENTRIES
}
```

The code for parsing the import table is basically the same as that for parsing the export table, and we can actually use the same function to parse entries in both tables. See the full code at the end of the chapter.

6.2.4 Code Chunk

The chunk named “Code” contains the beam code for the module and is mandatory. The format of the chunk is:

```
{“Code”:4
  CHUNKSIZE:4
  SUBSIZE:4
  INSTRUCTIONSET:4
  OPCODEMAX:4
  NUMBEROFLABELS:4
  NUMBEROFFUNCTIONS:4
  [OPCODE:1]:(CHUNKSIZE-SUBSIZE)
  [4-BYTEPAD:1]:0..3
}
```

The field SUBSIZE stores the number of words before the code starts. This makes it possible to add new information fields in the code chunk without breaking older loaders. The INSTRUCTIONSET field indicates which version of the instruction set the file uses. The version number is increased if any instruction is changed in an incompatible way.

The OPCODEMAX field indicates the highest number of any opcode used in the code. New instructions can be added to the system in a way such that older loaders still can load a newer file as long as the instructions used in the file are within the range the loader knows about.

The field NUMBEROFLABELS contains the number of labels so that a loader can preallocate a label table of the right size. The field NUMBEROFFUNCTIONS contains the number of functions so that a loader can preallocate a functions table of the right size.

We can parse out the code chunk by adding the following code to our program:

```
parse_chunks([{"Code", Size, <<SubSize:32/integer,Chunk/binary>>
              } | Rest], Acc) ->
   <<Info:SubSize/binary, Code/binary>> = Chunk,
   %% 8 is size of CunkSize & SubSize
   OpcodeSize = Size - SubSize - 8,
   <<OpCodes:OpcodeSize/binary, _Align/binary>> = Code,
   parse_chunks(Rest,[{code,parse_code_info(Info), OpCodes}
                      | Acc]);

..

parse_code_info(<<Instructionset:32/integer,
                  OpcodeMax:32/integer,
                  NumberOfLabels:32/integer,
                  NumberOfFunctions:32/integer,
                  Rest/binary>>) ->
   [{instructionset, Instructionset},
    {opcodemax, OpcodeMax},
    {numberoflabels, NumberOfLabels},
    {numberofFunctions, NumberOfFunctions} |
    case Rest of
         <<>> -> [];
         _ -> [{newinfo, Rest}]
    end].
```

We will learn how to decode the beam instructions in a later chapter, aptly named “BEAM Instructions”.

6.2.5 String table chunk

The chunk named “StrT” is mandatory and contains all constant string literals in the module as one long string. If there are no string literals the chunks should still be present but empty and of size 0.

The format of the chunk is:

```
{“StrT”:4
 CHUNKSIZE:4
 [STRINGBYTE]:CHUNKSIZE
 [4-BYTEPAD:1]:0..3
}
```

The string chunk can be parsed easily by just turning the string of bytes into a binary:

```
parse_chunks([{"StrT", _Size, <<Strings/binary>>} | Rest], Acc) ->
    parse_chunks(Rest,[{strings,binary_to_list(Strings)}|Acc]);
```

6.2.6 Attributes Chunk

The chunk named "Attr" is optional, but some OTP tools expect the attributes to be present. The releashandler expect the "vsn" attribute to be present. You can get the version attribute from a file with: beam_lib:version(Filename), this function assumes that there is an attribute chunk with a "vsn" attribute present.

The format of the chunk is:

```
{"Attr":4
  CHUNKSIZE:4
  ATTRIBUTES:CHUNKSIZE
  [4-BYTEPAD]:0..3
}
```

We can parse the attribute chunk like this:

```
parse_chunks([{"Attr", Size, Chunk} | Rest], Acc) ->
    <<Bin:Size/binary, _Pad/binary>> = Chunk,
    Attribs = binary_to_term(Bin),
    parse_chunks(Rest,[{attributes,Attribs}|Acc]);
```

6.2.7 Compilation Information Chunk

The chunk named "CInf" is optional, but some OTP tools expect the information to be present.

The format of the chunk is:

```
{"CInf":4
  CHUNKSIZE:4
  INFORMATION:CHUNKSIZE
  [4-BYTEPAD]:0..3
}
```

We can parse the compilation information chunk like this:

```
parse_chunks([{"CInf", Size, Chunk} | Rest], Acc) ->
    <<Bin:Size/binary, _Pad/binary>> = Chunk,
    CInfo = binary_to_term(Bin),
    parse_chunks(Rest,[{compile_info,CInfo}|Acc]);
```

6.2.8 Local Function Table Chunk

The chunk named "LocT" is optional and intended for cross reference tools.

The format is the same as that of the export table:

```
{"LocT":4
  CHUNKSIZE:4
  NUMBEROFENTRIES:4
  [{FUNCTION:4,
    ARITY:4,
    LABEL:4
   }]:NUMBEROFENTRIES
}
```

The code for parsing the local function table is basically the same as that for parsing the export and the import table, and we can actually use the same function to parse entries in all tables. See the full code at the end of the chapter.

6.2.9 Literal Table Chunk

The chunk named "LitT" is optional and contains compressed code literals. The format of the chunk is:

```
{"LitT":4
  CHUNKSIZE:4
  COMPRESSEDTABLESIZE:4
  <NUMBEROFLITERALS:4,
    [{SIZE:4
      LITERAL:SIZE
     }]:NUMBEROFLITERALS
   }>:COMPRESSEDTABLESIZE
}
```

The whole table is compressed with zlib:compress/1, and can be uncompressed with zlib_uncompress/1.

We can parse the chunk like this

```
parse_chunks([{"LitT", _ChunkSize,
              <<_CompressedTableSize:32, Compressed/binary>>}
             | Rest], Acc) ->
    <<_NumLiterals:32,Table/binary>> = zlib:uncompress(Compressed),
    Literals = parse_literals(Table),
    parse_chunks(Rest,[{literals,Literals}|Acc]);
```

...

```
parse_literals(<<Size:32,Literal:Size/binary,Tail/binary>>) ->
    [binary_to_term(Literal) | parse_literals(Tail)];
parse_literals(<<>>) -> [].
```

6.2.10 Abstract Code Chunk

The chunk named "Abst" is optional and may contain the code in abstract form. If you give the flag debug_info to the compiler it will store the abstract syntax tree for the module in this chunk. OTP tools like the debugger and Xref need the abstract form. The format of the chunk is:

```
{"Abst":4
  CHUNKSIZE:4
  [ABSTRACTCODE]:CHUNKSIZE
  [4-BYTEPAD:1]:0..3
}
```

We can parse the chunk like this

```
parse_chunks([{"Abst", _ChunkSize, <<>>} | Rest], Acc) ->
    parse_chunks(Rest,Acc);
parse_chunks([{"Abst", _ChunkSize, <<AbstractCode/binary>>} | Rest], Acc) ->
    parse_chunks(Rest,[{abstract_code,binary_to_term(AbstractCode)}|Acc]);
```

6.2.11 Compression and Encryption

TODO

6.2.12 Function Trace Chunk (Obsolete)

This chunk type is now obsolete.

6.2.13 Bringing it all Together

TODO

Chapter 7

Generic BEAM Instructions (25p)

Beam has two different instructions sets, an internal instructions set, called *specific*, and an external instruction set, called *generic*.

The generic instruction set is what could be called the official instruction set, this is the set of instructions used by both the compiler and the Beam interpreter. If there was an official Erlang Virtual Machine specification it would specify this instruction set. If you want to write your own compiler to the Beam, this is the instruction set you should target. If you want to write your own EVM this is the instruction set you should handle.

The external instruction set is quite stable, but it does change between Erlang versions, especially between major versions.

This is the instruction set which we will cover in this chapter.

The other instruction set, the specific, is an optimized instruction set used by the Beam to implement the external instruction set. To give you an understanding of how the Beam works we will cover this instruction set in Chapter 10. The internal instruction set can change without warning between minor version or even in patch releases. Basing any tool on the internal instruction set is risky.

In this chapter I will go through the general syntax for the instructions and some instruction groups in detail, a complete list of instructions with short descriptions can be found in Appendix .1.

7.1 Instruction definitions

The names and opcodes of the generic instructions are defined in `lib/compiler/src/genop.tab`.

The file contains a version number for the Beam instruction format, which also is wirtten to `.beam` files. This number has so far never changed and is still at version 0. If the external format would be changed in a non backwards compatible way this number would be changed.

The file genop.tab is used as input by `beam_makeops` which is a perl script which generate code from the ops tabs. The generator is used both to generate Erlang code for the compiler (beam_opcodes.hrl and beam_opcodes.erl) and to generate C code for the emulator (TODO: Filenames).

Any line in the file starting with "#" is a comment and ignored by `beam_makeops`. The file can contain definitions, which turns into a binding in the perl script, of the form:

```
NAME=EXPR
```

Like, e.g.:

```
BEAM_FORMAT_NUMBER=0
```

The Beam format number is the same as the `instructionset` field in the external beam format. It is only bumped when a backwards icnompatible change to the instruction set is made.

The main content of the file are the opcode definitions of the form:

```
OPNUM: [-]NAME/ARITY
```

Where OPNUM and ARITY are integers, NAME is an identifier starting with a lowercase letter (a-z), and :, -, and / are litterals.

For example:

```
1: label/1
```

The minus sign (-) indicates a depricated function. A depricated function keeps its opcode in order for the loader to be sort of backwards compatible (it will recognize depricated instructions and refuse to load the code).

In the rest of this Chapter we will go through some BEAM instructions in detail. For a full list with brief descriptions see: Appendix .1.

7.2 BEAM code listings

As we saw in Chapter 2 we can give the option *S* to the Erlang compiler to get a `.S` file with the BEAM code for the module in a human and machine readable format (actually as Erlang terms).

Given the file beamexample1.erl: -module(beamexample1). -export([id/1]). id(I) when is_integer(I) -> I.

When compiled with `erlc -S beamexample.erl` we get the following beamexmaple.S file: {module, beamexample1}. %% version = 0 {exports, [{id,1},{module_info,0},{module_info,1}]}. {attributes, []}. {labels, 7}. {function, id, 1, 2}. {label,1}. {line,[{location,"beamexample1.erl",5}]}. {func_info,{atom,beamexample1},{atom,id},1}. {label,2}. {test,is_integer,{f,1},[{x return. {function, module_info, 0, 4}. {label,3}. {line,[]}. {func_info,{atom,beamexample1},{atom,module_info},0}. {label,4}. {move,{atom,beamexample1},{x,0}}. {line,[]}. {call_ext_only,1,{extfunc,erlang,get_module_info,1}}. {function, module_info, 1, 6}. {label,5}. {line,[]}. {func_info,{atom,beamexample1},{atom,module_info},1}. {label,6}. {move,{x,0},{x,1}}. {move,{atom,beamexample1},{x,0}}. {line,[]}. {call_ext_only,2,{extfunc,erlang,get_module_info,2}}.

In addition to the actual beam code for the integer identity function we also get some meta instructions.

The first line `{module, beamexample1}.%% version =0` tells us the module name "beamexample1" and the version number for the instruction set "0".

Then we get a list of exported functions "id/1, module_info/0, module_info/1". As we can see the compiler has added two auto generated functions to the code. These two functions are just dispatchers to the generic module info BIFs (erlang:module_info/1 and erlang:module_info/2) with the name of the module added as the first argument.

The line {attributes, []} list all defined compiler attributes, none in our case.

Then we get to know that there are less than 7 labels in the module, `{labels, 7}`, which makes it easy to do code loading in one pass.

The last type of meta instruction is the `function` instruction on the format `{function, Name, Arity, StartLabel}`. As we can see with the `id` function the start label is actually the second label in the code of the function.

The instruction `{label, N}` is not really an instruction, it does not take up any space in memory when loaded. It is just to give a local name (or number) to a position in the code. Each label potentially marks the beginning of a basic block since it is a potential destination of a jump.

The first two instructions following the first label (`{label,1}`) are actually error generating code which adds the line number and module, function and arity information and throws an exception. That are the instructions `line` and `func_info`.

The meat of the function is after `{label,2}`, the instruction `{test,is_integer,{f,1},[{x,0}]}`. The test instruction tests if its arguments (in the list at the end, that is variable {x,0} in this case) fulfills the test, in this case is an integer (is_integer). If the test succeeds the next instruction (`return`) is executed. Otherwise the functions fails to label 1 (`{f,1}`), that is, execution continues at label one where a function clause exception is thrown.

The other two functions in the file are auto generated. If we look at the second function the instruction `{move,{x,0},{x,1}}` moves the argument in register x0 to the second argument register x1. Then the instruction `{move,{atom,beamexample1},{x,0}}` moves the module name atom to the first argument register x1. Finally a tail call is made to `erlang:get_module_info/2` (`{call_ext_only,2,{extfunc,erlang,get_module_info,2}}`). As we will see in the next section there are several different call instructions.

7.3 Calls

As we have seen in Chapter 8 there are several different types of calls in Erlang. To distinguish between local and remote calls in the instruction set, remote calls have `_ext` in their instruction names. Local calls just have a label in the code of the module, while remote calls takes a destination of the form `{extfunc, Module, Function, Arity}`.

To distinguish between ordinary (stack building) calls and tail-recursive calls, the latter have either `_only` or `_last` in their name. The variant with `_last` will also deallocate as many stack slot as given by the last argument.

There is also a `call_fun Arity` instruction that calls the closure stored in register {x, Arity}. The arguments are stored in x0 to {x, Arity-1}.

For a full listing of all types of call instructions see Appendix .1.

7.4 Stack (and Heap) Management

The stack and the heap of an Erlang process on Beam share the same memory area see Chapter 3 and Chapter 12 for a full discussion. The stack grows toward lower addresses and the heap toward higher addresses. Beam will do a garbage collection if more space than what is available is needed on either the stack or the heap.

A leaf function A leaf function is a function which doesn't call any other function.

A non leaf function A non leaf function is a function which may call another function.

On entry to a non leaf function the *continuation pointer* (CP) is saved on the stack, and on exit it is read back from the stack. This is done by the `allocate` and `deallocate` instructions, which are used for setting up and tearing down the stack frame for the current instruction.

A function skeleton for a leaf function looks like this: {function, Name, Arity, StartLabel}. {label,L1}. {func_info,{atom,Module},{aton
{label,L2}. ... return.

A function skeleton for a non leaf function looks like this: {function, Name, Arity, StartLabel}. {label,L1}. {func_info,{atom,Module},{
{label,L2}. {allocate,Need,Live}. ... call ... ... {deallocate,Need}. return.

The instruction `allocate StackNeed Live` saves the continuation pointer (CP) and allocate space for `StackNeed` extra words on the stack. If a GC is needed during allocation save `Live` number of X registers. E.g. if `Live` is 2 then registers X0 and X1 are saved.

When allocating on the stack, the stack pointer (E) is decreased. Before After | xxx | | xxx | E -> | xxx | | xxx | | | | ??? | caller save slot ... E -> | CP | ... ... HTOP -> | | HTOP -> | | | xxx | | xxx |

For a full listing of all types of allocate and deallocate instructions see Appendix .1.

7.5 Message Passing

Sending a message is straight forward in beam code. You just use the `send` instruction. Note though that the send instruction does not take any arguments, it is more like a function call. It assumes that the arguments (the destination and the message) are in the argument registers X0 and X1. The message is also copied from X1 to X0.

Receiving a message is a bit more complicated since it involves both selective receive with pattern matching and introduces a yield/resume point within a function body. (There is also a special feature to minimize message queue scanning using refs, more on that later.)

7.5.1 A Minimal Receive Loop

A minimal receive loop, which accepts any message and has no timeout (e.g. `receive _ -> ok end`) looks like this in BEAM code:

```
L1: loop_rec L2 x0
    remove_message
    ...
    jump L3

L2: wait L1

L3: ...
```

Figure 7.1: A simple receive loop.

The `loop_rec L2 x0` instruction first checks if there is any message in the message queue. If there are no messages execution jumps to L2, where the process will be suspended waiting for a message to arrive.

If there is a message in the message queue the `loop_rec` instruction also moves the message from the *m-buf* to the process heap. See Chapter 12 and Chapter 3 for details of the m-buf handling.

For code like `receive _ -> ok end`, where we accept any messages, there is no pattern matching needed, we just do a `remove_message` which unlinks the next message from the message queue and stores a pointer in X0. (It also removes any timeout, more on this soon.)

7.5.2 A Selective Receive Loop

For a selective receive like e.g. `receive [] -> ok end` we will loop over the message queue to check if any message in the queue matches.

```
L2: loop_rec L4  X0
    test is_nil L3 X0
    remove_message
    move {atom,ok} X0
    return
L3: loop_rec_end L2
L4: wait L2
```

Figure 7.2: A selective receive loop.

In this case we do a pattern match for Nil after the loop_rec instruction if there was a message in the mailbox. If the message doesn't match we end up at L3 where the instruction `loop_rec_end` advances the save pointer to the next message (`p->msg.save =&(*p->msg.save)->next`) and jumps back to L2.

If there are no more messages in the message queue the process is suspended by the `wait` instruction at L4 with the save pointer pointing to the end of the message queue. When the processes is rescheduled it will only look at new messages in the message queue (after the save point).

7.5.3 A Receive Loop With a Timeout

If we add a timeout to our selective receive the wait instruction is replaced by a wait_timeout instruction followed by a timeout instruction and the code following the timeout.

```
L6: loop_rec L8 X0
    test is_nil L7 X0
    remove_message
    move {atom,ok} X0
    return
L7: loop_rec_end L6
L8: wait_timeout L6 {integer,1000}
    timeout
    move {atom,done} X0
    return
```

Figure 7.3: A receive loop with a timeout.

The `wait_timeout` instructions sets up a timeout timer with the given time (1000 ms in our example) and it also saves the address of the next instruction (the `timeout`) in `p→def_arg_reg[0]` and then when the timer is set, `p→i` is set to point to def_arg_reg.

This means that if no matching message arrives while the process is suspended a timeout will be triggered after 1 second and execution for the process will continue at the timeout instruction.

Note that if a message that doesn't match arrives in the mailbox, the process is scheduled for execution and will run the pattern matching code in the receive loop, but the timeout will not be canceled. It is the `remove_message` code which also removes any timeout timer.

The `timeout` instruction resets the save point of the mailbox to the first element in the queue, and clears the timeout flag (F_TIMO) from the PCB.

7.5.4 The Synchronous Call Trick (aka The Ref Trick)

We have now come to the last version of our receive loop, where we use the ref trick alluded to earlier to avoid a long message box scan.

A common pattern in Erlang code is to implement a type of "remote call" with send and a receive between two processes. This is for example used by gen_server. This code is often hidden behind a library of ordinary function calls. E.g., you call the function `counter:increment(Counter)` and behind the scene this turns into something like `Counter ! {self(), inc}, receive {Counter, Count} → Count end`.

This is usually a nice abstraction to encapsulate state in a process. There is a slight problem though when the mailbox of the calling process has many messages in it. In this case the receive will have to check each message in the mailbox to find out that no message except the last matches the return message.

This can quite often happen if you have a server that receives many messages and for each message does a number of such remote calls, if there is no back throttle in place the servers message queue will fill up.

To remedy this there is a hack in ERTS to recognize this pattern and avoid scanning the whole message queue for the return message.

The compiler recognizes code that uses a newly created reference (ref) in a receive (see Figure 7.4), and emits code to avoid the long inbox scan since the new ref can not already be in the inbox.

```
Ref = make_ref(),
Counter ! {self(), inc, Ref},
receive
  {Ref, Count} -> Count
end.
```

Figure 7.4: The Ref Trick Pattern.

This gives us the following skeleton for a complete receive, see Figure 7.5.

```
     ...
     recv_mark L11
     call_ext 0 {extfunc,erlang,make_ref,0}
     ...

     recv_set L11
L11: loop_rec L13 x0
     test is_eq_exact L12 [X0, Y0]
     remove_message
     ...

L12: loop_rec_end L11
L13: wait_timeout L11 {integer,1000}
     timeout.
     ...
```

Figure 7.5: A receive with the ref_trick.

The `recv_mark` instruction saves the current position (the end `msg.last`) in `msg.saved_last` and the address of the label in `msg.mark`

The `recv_set` instruction checks that `msg.mark` points to the next instruction and in that case moves the save point (`msg.save`) to the last message received before the creation of the ref (`msg.saved_last`). If the mark is invalid (i.e. not equal to `msg.save`) the instruction does nothing.

Chapter 8

Different Types of Calls, Linking and Hot Code Loading (5p)

Local calls, remote calls, closure calls, tuple calls, p-mod calls. The code server. Linking. Hot code loading, purging. (Overlap with Chapter 4, have to see what goes where.) Higher order functions, Implementation of higher order functions. Higher order functions and hot code loading. Higher order functions in a distributed system.

8.1 Hot Code Loading

In Erlang there is a semantic difference between a local function call and a remote function call. A remote call, that is a call to a function in a named module, is guaranteed to go to the latest loaded version of that module. A local call, a unqualified call to a function within the same module, is guaranteed to go to the same version of the code as the caller.

A call to a local function can be turned into a remote call by specifying the module name at the call site. This is usually done with the ?MODULE macro as in `?MODULE:foo()`. A remote call to a non local module can not be turned into a local call, i.e. there is no way to guarantee the version of the callee in the caller.

This is an important feature of Erlang which makes *hot code loading* or *hot upgrades* possible. Just make sure you have a remote call somewhere in your server loop and you can then load new code into the system while it is running; when execution reaches the remote call it will switch to executing the new code.

A common way of writing server loops is to have a local call for the main loop and a code upgrade handler which does a remote call and possibly a state upgrade:

```
loop(State) ->
  receive
    upgrade ->
       NewState = ?MODULE:code_upgrade(State),
       ?MODULE:loop(NewState);
     Msg ->
       NewState = handle_msg(Msg, State),
       loop(NewState)
   end.
```

With this construct, which is basically what gen_server uses, the programmer has control over when and how a code upgrade is done.

The hot code upgrade is one of the most important features of Erlang which makes it possible to write servers that operates 24/7 year out and year in. It is also one of the main reasons why Erlang is dynamically typed. It is very hard in a statically typed language to give type for the code_upgrade function. (It is also hard to give the type of the loop function) These types will change in the future as the type of State changes to handle new features.

For a language implementer concerned with performance, the hot code loading functionality is a burden though. Since each call to or from a remote module can change to new code in the future it is very hard to do whole program optimization across module boundaries. (Hard but not impossible, there are solutions but so far I have not seen one fully implemented.)

8.2 Code Loading

In the Erlang Runtime System the code loading is handled by the code server. The code server will call the lover level bifs in the `erlang` module for the actual loading. But the code server also determines the purging policy.

The runtime system can keep two versions of each module, a *current* version and an *old* version. All fully qualified (remote) calls goes to the current version. Local calls in the old version and return addresses on the stack can still go to the old version.

If a third version of a module is loaded and there still are processes running (have pointers on the stack to) old code the code server will kill does processes and purge the old code. Then the current version will become old and the third version will be loaded as the current version.

8.3 Transforming from Generic to Specific instructions

The BEAM loader does not just take the external beam format and writes it to memory. It also does a number of transformations on the code and translates from the external (generic) format to the internal (specific) format.

The code for the loader can be found in `beam_load.c` (in `erts/emulator/`) but most of the logic for the translations are in the file `ops.tab` (in the same directory).

The first step of the loader is to parse beam file, basically the same work as we did in Erlang in Chapter 6 but written in C.

Then the rules in ops.tab are applied to instructions in the code chunk to translate the generic instruction to one or more specific instructions.

The translation table works through pattern matching. Each line in the file defines a pattern of one or more generic instructions with arguments and optionally an arrow followed by one or more instructions to translate to.

The transformations in ops tab tries to handle patterns of instructions generated by the compiler and peephole optimize them to fewer specific instructions. The ops tab transformations tries to generate jump tables for patterns of selects.

The file ops.tab is not parsed at runtime, instead a pattern matching program is generated from ops.tab and stored in an array in a generated C file. The perl script `beam_makeops` (in `erts/emulator/utils`) generates a target specific set of opcodes and translation programs in the files `beam_opcodes.h` and `beam_opcodes.c` (these files end up in the given target directory e.g. `erts/emulator/x86_64-unknown-linux-gnu/opt/smp/`).

The same program (beam_makeops) also generates the Erlang code for the compiler back end `beam_opcodes.erl`.

Chapter 9

The BEAM Loader

9.1 Transforming from Generic to Specific instructions

The BEAM loader does not just take the external beam format and writes it to memory. It also does a number of transformations on the code and translates from the external (generic) format to the internal (specific) format.

The code for the loader can be found in `beam_load.c` (in `erts/emulator/`) but most of the logic for the translations are in the file `ops.tab` (in the same directory).

The first step of the loader is to parse beam file, basically the same work as we did in Erlang in Chapter 6 but written in C.

Then the rules in ops.tab are applied to instructions in the code chunk to translate the generic instruction to one or more specific instructions.

The translation table works through pattern matching. Each line in the file defines a pattern of one or more generic instructions with arguments and optionally an arrow followed by one or more instructions to translate to.

The transformations in ops tab tries to handle patterns of instructions generated by the compiler and peephole optimize them to fewer specific instructions. The ops tab transformations tries to generate jump tables for patterns of selects.

The file ops.tab is not parsed at runtime, instead a pattern matching program is generated from ops.tab and stored in an array in a generated C file. The perl script `beam_makeops` (in `erts/emulator/utils`) generates a target specific set of opcodes and translation programs in the files `beam_opcodes.h` and `beam_opcodes.c` (these files end up in the given target directory e.g. `erts/emulator/x86_64-unknown-linux-gnu/opt/smp/`).

The same program (beam_makeops) also generates the Erlang code for the compiler back end `beam_opcodes.erl`.

Chapter 10

BEAM Internal Instructions

Chapter 11

Scheduling

To fully understand where time in an ERTS system is spent you need to understand how the system decides which Erlang code to run and when to run it. These decisions are made by the Scheduler.

The scheduler is responsible for the real-time guarantees of the system. In a strict Computer Science definition of the word real-time, a real-time system have to be able to guarantee a response within a specified time. That is, there are real deadlines and each task has to complete before its deadline. In Erlang there are no such guarantees, a *timeout* in Erlang is only guaranteed to **not** trigger **before** the given deadline.

In a general system like Erlang where we want to be able to handle all sorts of programs and loads, the scheduler will have to make some compromises. There will always be corner cases where a generic scheduler will behave badly. After reading this chapter you will have a deeper understanding of how the Erlang scheduler works and especially when it might not work optimally. You should be able to design your system to avoid the corner cases and you should also be able to analyze a misbehaving system.

11.1 Concurrency, Parallelism, and Preemptive Multitasking

Erlang is a concurrent language. When we say that processes run concurrently we mean that for an outside observer it looks like two processes are executing at the same time. In a single core system this is achieved by preemptive multitasking. This means that one process will run for a while, and then the scheduler of the virtual machine will suspend it and let another process run.

In a multicore or a distributed system we can achieve true parallelism, that is, two or more processes actually executing at the exact same time. In an SMP enabled emulator the system uses several OS threads to indirectly execute Erlang processes by running one scheduler and emulator per thread. In a system using the default settings for ERTS there will be one thread per enabled core (physical or hyper threaded).

We can check that we have a system capable of parallel execution, by checking if SMP support is enabled:

```
iex(1)> :erlang.system_info :smp_support
true
```

We can also check how many schedulers we have running in the system:

```
iex(2)> :erlang.system_info :schedulers_online
4
```

We can see this information in the Observer as shown in the figure below.

If we spawn more processes than we have have schedulers and let them do some busy work we can see that there are a number of processes *running* in parallel and some processes that are *runnable* but not currently running. We can see this with the function `erlang:process_info/2`.

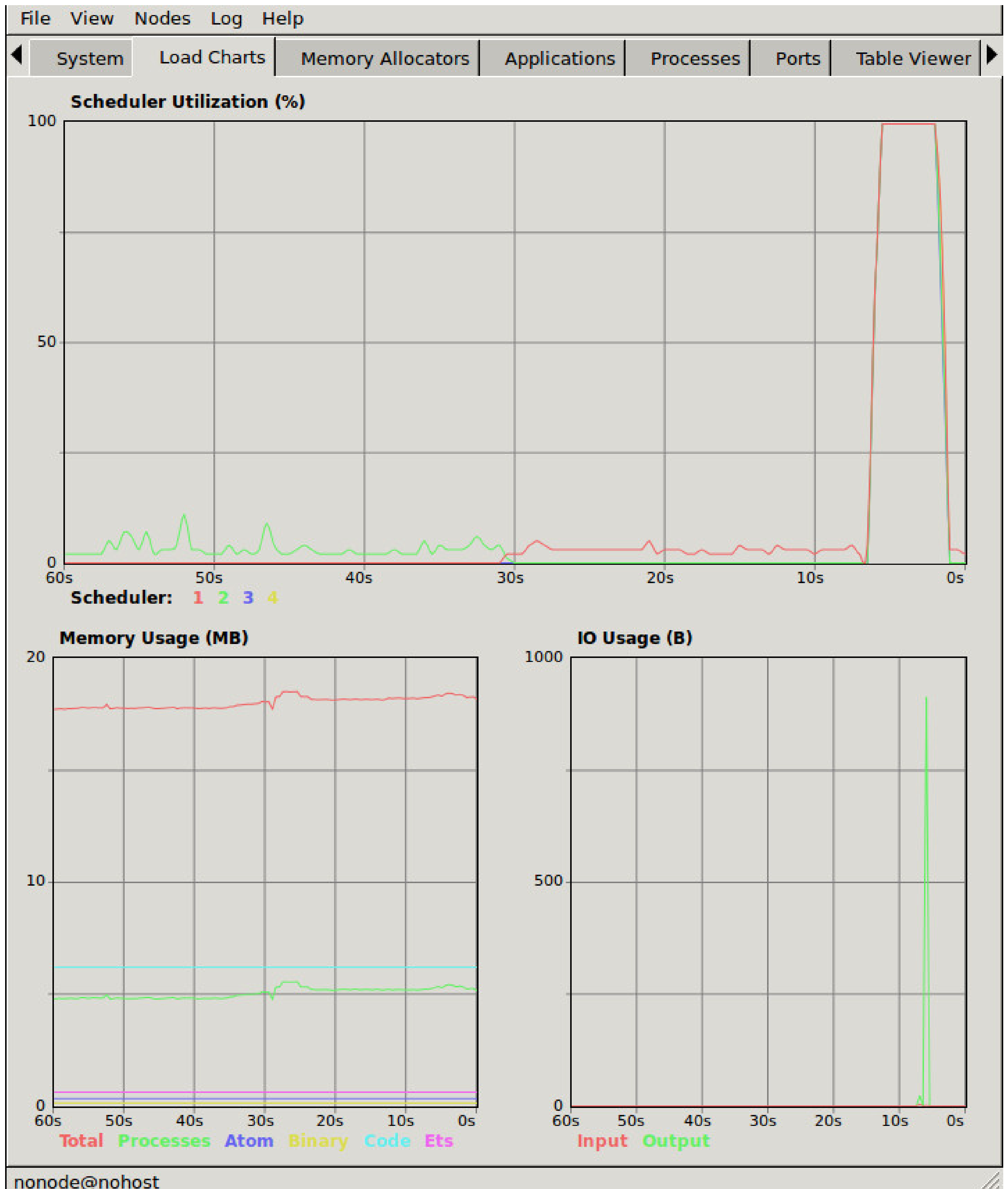
```
1> Loop = fun (0, _) -> ok; (N, F) -> F(N-1, F) end,
   BusyFun = fun() -> spawn(fun () -> Loop(1000000, Loop) end) end,
   SpawnThem = fun(N) -> [ BusyFun() || _ <- lists:seq(1, N)] end,
```

```
   GetStatus = fun() -> lists:sort([{erlang:process_info(P, [status]), P}
                        || P <- erlang:processes()]) end,
   RunThem = fun (N) -> SpawnThem(N), GetStatus() end,
   RunThem(8).

[{[{status,garbage_collecting}],<0.62.0>},
 {[{status,garbage_collecting}],<0.66.0>},
 {[{status,runnable}],<0.60.0>},
 {[{status,runnable}],<0.61.0>},
 {[{status,runnable}],<0.63.0>},
 {[{status,runnable}],<0.65.0>},
 {[{status,runnable}],<0.67.0>},
 {[{status,running}],<0.58.0>},
 {[{status,running}],<0.64.0>},
 {[{status,waiting}],<0.0.0>},
 {[{status,waiting}],<0.1.0>},

...
```

We will look closer at the different statuses that a process can have later in this chapter, but for now all we need to know is that a process that is *running* or *garbage_collecting* is actually running in on a scheduler. Since the machine in the example has four cores and four schedulers there are four process running in parallel (the shell process and three of the *busy processes*). There are also five busy processes waiting to run in the state *runnable*.

By using the *Load Charts* tab in the Observer we can see that all four schedulers are fully loaded while the busy processes execute.

```
observer:start().
ok
3> RunThem(8).
```



11.2 Preemptive Multitasking in ERTS Cooperating in C

The preemptive multitasking on the Erlang level is achieved by cooperative multitasking on the C level. The Erlang language, the compiler and the virtual machine works together to ensure that the execution of an Erlang process will yield within a limited

time and let the net process run. The technique used to measure and limit the allowed execution time is called reduction counting, we will look at all the details of reduction counting soon.

11.3 Reductions

One can describe the scheduling in BEAM as preemptive scheduling on top of cooperative scheduling. A process can only be suspended at certain points of the execution, such as at a receive or a function call. In that way the scheduling is cooperative---a process has to execute code which allows for suspension. The nature of Erlang code makes it almost impossible for a process to run for a long time without doing a function call. There are a few Built In Functions (BIFs) that still can take too long without yielding. Also, if you call C code in a badly implemented Native Implemented Function (NIF) you might block one scheduler for a long time. We will look at how to write well behaved NIFs in <ref linkend="ch.c"/>.)

Since there are no other loop constructs than recursion and list comprehensions, there is no way to loop forever without doing a function call. Each function call is counted as a `reduction`; when the reduction limit for the process is reached it is suspended.

Note
Reductions The term reduction comes from the Prolog ancestry of Erlang. In Prolog each execution step is a goal-reduction, where each step reduces a logic problem into its constituent parts, and then tries to solve each part.

11.3.1 How Many Reductions Will You Get?

When a process is scheduled it will get a number of reductions defined by `CONTEXT_REDS` (defined in `erl_vm.h`, currently as 2000). After using up its reductions or when doing a receive without a matching message in the inbox, the process will be suspended and a new processes will be scheduled.

If the VM has executed as many reductions as defined by `INPUT_REDUCTIONS` (currently `2*CONTEXT_REDS`, also defined in `erl_vm.h`) or if there is no process ready to run the scheduler will do system-level activities. That is, basically, check for IO; we will cover the details soon.

11.3.2 What is a Reduction Really?

It is not completely defined what a reduction is, but at least each function call should be counted as a reduction. Things get a bit more complicated when talking about BIFs and NIFs. A process should not be able to run for "a long time" without using a reduction and yielding. A function written in C can not yield in the middle, it has to make sure it is in a clean state and return. In order to be re-entrant it has to save its internal state somehow before it returns and then set up the state again on re-entry. This can be very costly, especially for a function that sometimes only does little work and sometimes lot. The reason for writing a function in C instead of Erlang is usually to achieve performance and to not do unnecessary book keeping work. Since there is no clear definition of what one reduction is, other than a function call on the Erlang level, there is a risk that a function implemented in C takes many more clock cycles per reduction than a normal Erlang function. This can lead to an imbalance in the scheduler, and even starvation.

For example in Erlang versions prior to R16, the BIFs `binary_to_term/1` and `term_to_binary/1` were non yielding and only counted as one reduction. This meant that a process calling these functions on large terms could starve other processes. This can even happen in a SMP system because of the way processes are balanced between schedulers, which we will get to soon.

While a process is running the emulator keeps the number of reductions left to execute in the (register mapped) variable `FCALLS` (see `beam_emu.c`).

We can examine this value with `hipe_bifs:show_pcb/1`: iex(13)> :hipe_bifs.show_pcb self P: 0x00007efd7c2c0400 ----- -- Offset| Name | Value | *Value | 0 | id | 0x00000270000004e3 | | ... 328 | rcount | 0x0000000000000000 | | 336 | reds | 0x000000000000a528 | | ... 320 | fcalls | 0x00000000000004a3 | |

The field `reds` keep track of the total number of reductions a process has done up until it was last suspended. By monitoring this number you can see which processes do the most work.

You can see the total number of reductions for a process (the reds field) by calling `erlang:process_info/2` with the atom `reductions` as the second argument. You can also see this number in the process tab in the observer or with the i/0 command in the Erlang shell.

As noted earlier, each time a process starts the field fcalls is set to the value of `CONTEXT_REDS` and for each function call the process executes fcalls is reduced by 1. When the process is suspended the field reds is increased by the number of executed reductions. In some C like code something like: `p->reds += (CONTEXT_REDS - p->fcalls).`

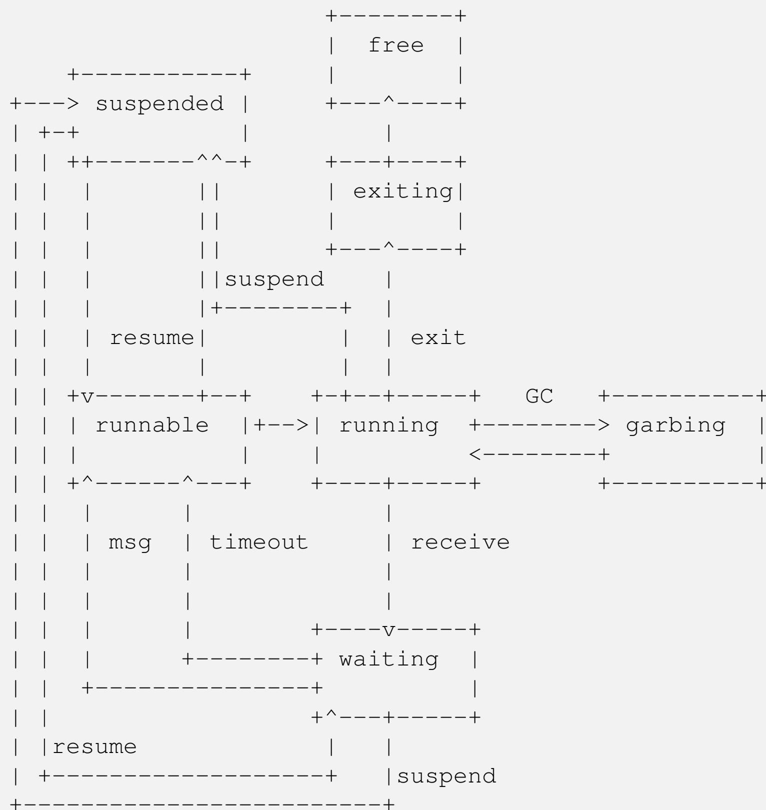
Normally a process would do all its alloted reductions and `fcalls` would be 0 at this point, but if the process suspends in a receive waiting for a message it will have some reductions left.

When a process uses up all its reductions it will yield to let another process run, it will go from the process state *running* to the state *runnable*, if it yields in a receive it will instead go into the state *waiting* (for a message). In the next section we will take a look at all the different states a process can be in.

11.4 The Process State (or *status*)

The field `status` in the PCB contains the process state. It can be one of *free*, *runnable*, *waiting*, *running*, *exiting*, *garbing*, and *suspended*. When a process exits it is marked as free---you should never be able to see a process in this state, it is a short lived state where the process no longer exist as far as the rest of the system is concerned but there is still some clean up to be done (freeing memory and other resources).

Each process status represents a state in the Process State Machine. Events such as a timeout or a delivered message triggers transitions along the edges in the state machine. The *Process State Machine* looks like this:



The normal states for a process are *runnable*, *waiting*, and *running*. A running process is currently executing code in one of the schedulers. When a process enters a receive and there is no matching message in the message queue, the process will become waiting until a message arrives or a timeout occurs. If a process uses up all its reductions, it will become runnable and wait for a scheduler to pick it up again. A waiting process receiving a message or a timeout will become runnable.

Whenever a process needs to do garbage collection, it will go into the *garbing* state until the GC is done. While it is doing GC is saves the old state in the field `gcstatus` and when it is done it sets the state back to the old state using `gcstatus`.

The suspended state is only supposed to be used for debugging purposes. You can call `erlang:suspend_process/2` on another process to force it into the suspended state. Each time a process calls `suspend_process` on another process, the *suspend count* is increased. This is recorded in the field `rcount`. A call to (`erlang:resume_process/1`) by the suspending process will decrease the suspend count. A process in the suspend state will not leave the suspend state until the suspend count reaches zero.

The field `rstatus` (resume status) is used to keep track of the state the process was in before a suspend. If it was *running* or *runnable* it will start up as *runnable*, and if it was *waiting* it will go back to the wait queue. If a suspended waiting process receives a timeout `rstatus` is set to *runnable* so it will resume as *runnable*.

To keep track of which process to run next the scheduler keeps the processes in a queue.

11.5 Process Queues

The main job of the scheduler is to keep track of work queues, that is, queues of processes and ports.

There are two process states that the scheduler has to handle, *runable*, and *waiting*. Processes waiting to receive a message are in the waiting state. When a waiting process receives a message the send operations triggers a move of the receiving process into the runable state. If the receive statement have a timeout the scheduler has to trigger the state transition to runable when the timeout triggers. We will cover this mechanism later in this chapter.

11.5.1 The Ready Queue

Processes in the runable state are placed in a FIFO (first in first out) queue handled by the scheduler, called the *ready queue*. The queue is implemented by a first and a last pointer and by the next pointer in the PCB of each participating process. When a new process is added to the queue the *last* pointer is followed and the process is added to the end of the queue in an O(1) operation. When a new process is scheduled it is just popped from the head (the *first* pointer) of the queue.

```
The Ready Queue

First: -->  P5       +---> P3       +-+-> P17
            next: ---+     next: ---+ |  next: NULL
                                      |
Last: --------------------------------+
```

In a SMP system, where you have several scheduler threads, there is one queue per scheduler.

```
Scheduler 1       Scheduler 2      Scheduler 3      Scheduler 4

Ready: P5         Ready: P1        Ready: P7        Ready: P9
       P3                P4               P12
       P17                                P10
```

The reality is slightly more complicated since Erlang processes have priorities. Each scheduler actually have three queues. One queue for *max priority* tasks, one for *high priority* tasks and one queue containing both *normal* and *low priority* tasks.

```
Scheduler 1       Scheduler 2      Scheduler 3      Scheduler 4

Max:    P5        Max:             Max:             Max:
High:             High:  P1        High:            High:
Normal: P3        Ready: P4        Ready: P7        Ready: P9
        P17                               P12
                                          P10
```

If there are any processes int the max queue the scheduler will pick these processes for execution. If there are no processes in the max queue but there are processes in the high priority queue the scheduler will pick those processes. Only if there are no processes in the max and the high priority queues will the scheduler pick the first process from the normal and low queue.

When a normal processes is inserted into the queue it get a *schedule count* of 1 and a low prio process get a schedule count of 8. When a process is picked from the front of the queue its schedule count is reduced by one, if the count reaches zero the process is scheduled, otherwise it is inserted at the end of the queue. This means that low priority processes will go through the queue seven times before they are scheduled.

11.5.2 Waiting, Timeouts and the Timing Wheel

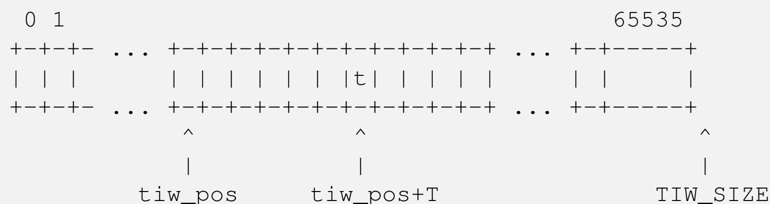
A processs trying to do a receive on an empty mailbox or on a mailbox with no matching messages will yield and go into the waiting state.

When a message is deliverd to an inbox the sending process will check wether the receiver is *sleeping* in the waiting state, and in that case it will *wake* the process, change its state to runable, and put it at the end of the appropriate ready queue.

If the receive statement has a `timeout` clause a timer will be created for the process which will trigger after the specified timeout time. The only guarantee the runtime system gives on a timeout is that it will not trigger before the set time, it might be some time after the intended time before the process is scheduled and get to execute.

Timers are handled in the VM by a *timing wheel*. That is, an array of time slots which wraps around. The timing wheel is a global resource and there might be contention for the write lock to the timing wheel if you have many processes inserting timers into the wheel.

The default size (`TIW_SIZE`) of the timing wheel is 65536 slots (or 8192 slots if you have built the system for a small memory footprint). The current time is indicated by an index into the array (`tiw_pos`). When a timer is inserted into the wheel with a timeout of T the timer is inserted into the slot at `(tiw_pos+T)%TIW_SIZE`.



```
 0 1                                      65535
+-+-+- ... +-+-+-+-+-+-+-+-+-+-+-+ ... +-+-----+
| | |      | | | | | | |t| | | | |     | |     |
+-+-+- ... +-+-+-+-+-+-+-+-+-+-+-+ ... +-+-----+
             ^           ^                  ^
             |           |                  |
          tiw_pos     tiw_pos+T          TIW_SIZE
```

The timer stored in the timing wheel is a pointer to an `ErlTimer` struct. See [erl_time.h](https://github.com/erlang/otp/blob/-OTP-19.1/erts/emulator/beam/erl_time.h). If several timers are inserted into the same slot they are linked together in a linked list by the `prev` and `next` fields. The `count` field is set to `T/TIW_SIZE`

```
/*
** Timer entry:
*/
typedef struct erl_timer {
    struct erl_timer* next;     /* next entry tiw slot or chain */
    struct erl_timer* prev;     /* prev entry tiw slot or chain */
    Uint slot;                  /* slot in timer wheel */
    Uint count;                 /* number of loops remaining */
    int    active;              /* 1=activated, 0=deactivated */
    /* called when timeout */
    void (*timeout)(void*);
    /* called when cancel (may be NULL) */
    void (*cancel)(void*);
    void* arg;         /* argument to timeout/cancel procs */
} ErlTimer;
```

11.6 Ports

A port is an Erlang abstraction for a communication point with the world outside of the Erlang VM. Communications with sockets, pipes, and file IO are all done through ports on the Erlang side.

A port, like a process, is created on the same scheduler as the creating process. Also like processes port uses reductions to decide when to yield, and they also get to run for 2000 reductions. But since ports don't run Erlang code there are no Erlang function calls to count as reductions, instead each *port task* is counted as a number of reductions. Currently a task uses a little more than 200 reductions per task, and a number of reductions relative to one thousands of the size of transmitted data.

A port task is one operation on a port, like opening, closing, sending a number of bytes or receiving data. In order to execute a port task the executing thread takes a lock on the port.

Port tasks are scheduled and executed in each iteration in the scheduler loop (see below) before a new process is selected for execution.

11.7 Reductions

When a process is scheduled it will get a number of reductions defined by CONTEXT_REDS (defined in erl_vm.h, currently as 2000). After using up its reductions or when doing a receive without a matching message in the inbox, the process will be suspended and a new processes will be scheduled.

If the VM has executed as many reductions as defined by INPUT_REDUCTIONS (currently 2*CONTEXT_REDS, also defined in `erl_vm.h`) or if there is no process ready to run the scheduler will do system-level activities. That is, basically, check for IO; we will cover the details soon.

It is not completely defined what a reduction is, but at least each function call should be counted as a reduction. Things get a bit more complicated when talking about BIFs and NIFs. A process should not be able to run for "a long time" without using a reduction and yielding. A function written in C can usually not yield at any time, and the reason for writing it in C is usually to achieve performance. In such functions a reduction might take longer which can lead to imbalance in the scheduler.

For example in Erlang versions prior to R16 the BIFs `binary_to_term/1` and `term_to_binary/1` where non yielding and only counted as one reduction. This meant that a process calling theses functions on large terms could starve other processes. This can even happen in a smp system because of the way processes are balanced between schedulers, which we will get to soon.

While a process is running the emulator keeps the number of reductions left to execute in the (register mapped) variable FCALLS (see `beam_emu.c`).

11.8 The Scheduler Loop

Conceptually you can look at the scheduler as the driver of program execution in the Erlang VM. In reality, that is, the way the C code is structured, it is the emulator (`process_main` in beam_emu.c) that drives the execution and it calls the scheduler as a subroutine to find the next process to execute.

Still, we will pretend that it is the other way around, since it makes a nice conceptual model for the scheduler loop. That is, we see it as the scheduler picking a process to execute and then handing over the execution to the emulator.

Looking at it that way, the scheduler loop looks like this:

1. Update reduction counters.
2. Check timers
3. If needed check balance
4. If needed migrate processes and ports
5. Do auxiliary scheduler work
6. If needed check IO and update time
7. While needed pick a port task to execute
8. Pick a process to execute

11.9 Load Balancing

The current strategy of the load balancer is to use as few schedulers as possible without overloading any CPU. The idea is that you will get better performance through better memory locality when processes share the same CPU.

One thing to note though is that the load balancing done in the scheduler is between scheduler threads and not necessarily between CPUs or cores. When you start the runtime system you can specify how schedulers should be allocated to cores. The default behaviour is that it is up to the OS to allocated scheduler threads to cores, but you can also choose to bind schedulers to cores.

The load balancer assumes that there is one schedulers running on each core so that moving a process from a overloaded scheduler to an under utilized scheduler will give you more parallel processing power. If you have changed how schedulers are allocated to cores, or if you OS is overloaded or bad at assigning threads to cores, the load balancing might actually work against you.

The load balancer uses two techniques to balance the load, *task stealing* and *migration*. Task stealing is used every time a scheduler runs out of work, this technique will result in the work becoming more spread out between schedulers. Migration is more complicated and tries to compact the load to the right number of schedulers.

11.9.1 Task Stealing

If a scheduler run queue is empty when it should pick a new process to schedule the scheduler will try to steal work from another scheduler.

First the scheduler takes a lock on itself to prevent other schedulers to try to steal work from the current scheduler. Then it checks if there are any inactive schedulers that it can steal a task from. If there are no inactive schedulers with stealable tasks then it will look at active schedulers, starting with schedulers having a higher id than itself, trying to find a stealable task.

The task stealing will look at one scheduler at a time and try to steal the highest priority task of that scheduler. Since this is done per scheduler there might actually be higher priority tasks that are stealable on another scheduler which will not be taken.

The task stealing tries to move tasks towards schedulers with lower numbers by trying to steal from schedulers with higher numbers, but since the stealing also will wrap around and steal from schedulers with lower numbers the result is that processes are spread out on all active schedulers.

Task stealing is quite fast and can be done on every iteration of the scheduler loop when a scheduler has run out of tasks.

11.9.2 Migration

To really utilize the schedulers optimally a more elaborate migration strategy is used. The current strategy is to compact the load to as few schedulers as possible, while at the same time spread it out so that no scheduler is overloaded.

This is done by the function *check_balance* in *erl_process.c*.

The migration is done by first setting up a migration plan and then letting schedulers execute on that plan until a new plan is set up. Every 2000*2000 reductions a scheduler calculates a migration path per priority per scheduler by looking at the workload of all schedulers. The migration path can have three different types of values: 1) cleared 2) migrate to schedueler # 3) immigrate from scheduler #

When a process becomes ready (for example by reciving a message or triggering a timeout) it will normally be scheduled on the last scheduler it ran on (S1). That is, if the migration path of that scheduler (S1), at that priority, is cleared. If the migration path of the scheduler is set to emigrate (to S2) the process will be handed over to that scheduler if both S1 and S2 have unbalanced run-queues. We will get back to what that means.

When a scheduler (S1) is to pick a new process to execute it checks to see if it has an immigration path from (S2) set. If the two involved schedulers have unbalanced run-queues S1 will steal a process from S2.

The migration path is calculated by comparing the maximum run-queues for each scheduler for a certain priority. Each scheduler will update a counter in each iteration of its scheduler loop keeping track of the maximal queue length. This information is then used to calculate an average (max) queue length (*AMQL*).

```
 Max
 Run Q
 Length
    5         o
              o
           o  o
Avg: 2.5 --------------
           o  o     o
    1      o  o     o

scheduler S1 S2 S3 S4
```

Then the schedulers are sorted on their max queue lengths.

```
 Max
 Run Q
 Length
    5               o
                    o
                 o  o
Avg: 2.5 --------------
              o  o  o
    1         o  o  o

scheduler S3 S4 S1 S2

           ^        ^
           |        |
          tix      fix
```

Any scheduler with a longer run queue than average (S1, S2) will be marked for emigration and any scheduler with a shorter max run queue than average (S3, S4) will be targeted for immigration.

This is done by looping over the ordered set of schedulers with two indices (immigrate from (`fix`)) and (emigrate to (`tix`)). In each iteration of the a loop the immigration path of S[tix] is set to S[fix] and the emigration path of S[fix] is set to S[tix]. Then tix is increased and fix decreased till they both pass the balance point. If one index reaches the balance point first it wraps.

In the example: * Iteration 1: S2.emigrate_to = S3 and S3.immigrate_from = S2 * Iteration 2: S1.emigrate_to = S4 and S4.immigrate_from = S1

Then we are done.

In reality things are a bit more complicated since schedulers can be taken off line. The migration planning is only done for online schedulers. Also, as mentioned before, this is done per priority level.

When a process is to be inserted into a ready queue and there is a migration path set from S1 to S2 the scheduler first checks that the run queue of S1 is larger than AMQL and that the run queue of S2 is smaller than the average. This way the migration is only allowed if both queues are still unbalanced.

There are two exception though where a migration is forced even when the queues are balanced or even imbalanced in the wrong way. In both these cases a special evacuation flag is set which overrides the balance test.

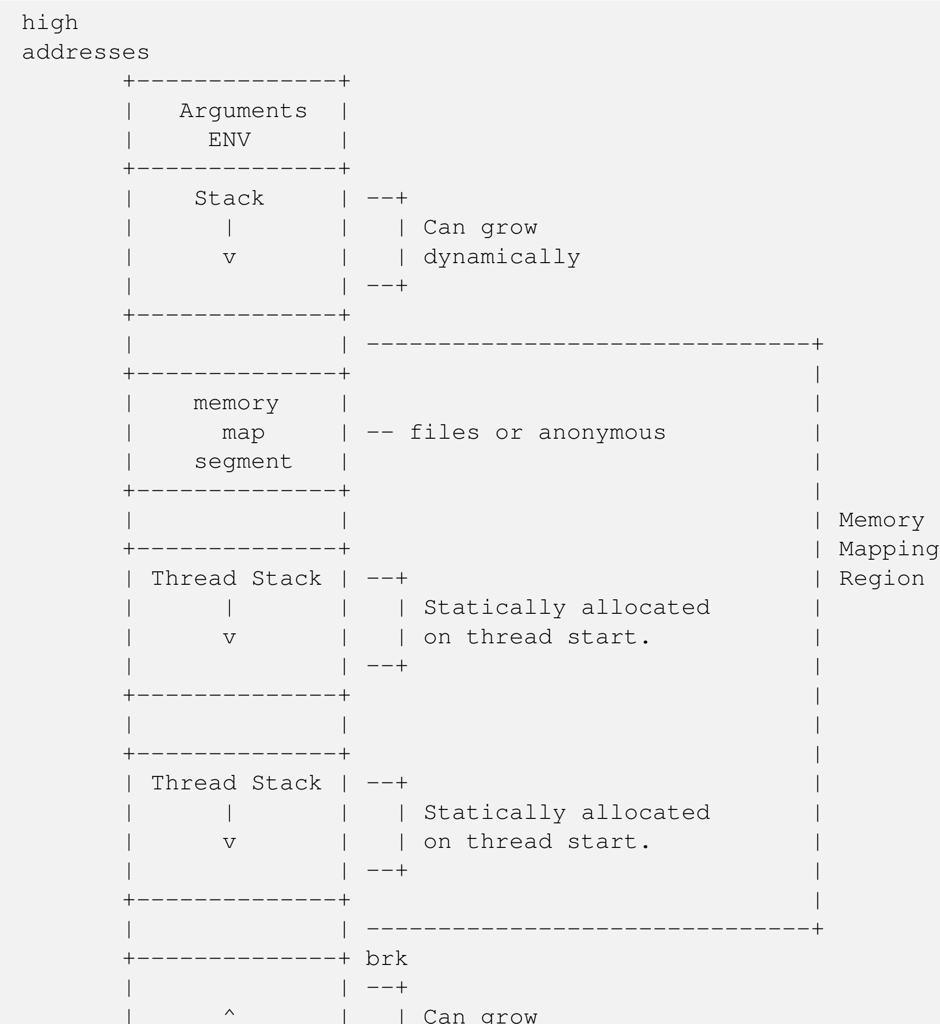
The evacuation flag is set when a scheduler is taken off line to ensure that no new processes are scheduled on an off line scheduler. The flag is also set when the scheduler detects that no progress is made on some priority. That is, if there for example is a max priority process which always is ready to run so that no normal priority processes ever are scheduled. Then the evacuation flag will be set for the normal priority queue for that scheduler.

Chapter 12

The Memory Subsystem: Stacks, Heaps and Garbag Collection

Before we dive into the memory subsystem of ERTS, we need to have some basic vocabulary and understanding of the general memory layout of a program in a modern operating system. In this review section I will assume the program is compiled to an ELF executable and running on Linux on something like an IA-32/AMD64 architecture. The layout and terminology is basically the same for all operating systems that ERTS compile on.

A program's memory layout looks something like this:



```
high
addresses
       +--------------+
       |   Arguments  |
       |     ENV      |
       +--------------+
       |    Stack     | --+
       |      |       |   | Can grow
       |      v       |   | dynamically
       |              | --+
       +--------------+
       |              | -------------------------------+
       +--------------+                                |
       |    memory    |                                |
       |      map     | -- files or anonymous          |
       |    segment   |                                |
       +--------------+                                |
       |              |                                | Memory
       +--------------+                                | Mapping
       | Thread Stack | --+                            | Region
       |      |       |   | Statically allocated       |
       |      v       |   | on thread start.           |
       |              | --+                            |
       +--------------+                                |
       |              |                                |
       +--------------+                                |
       | Thread Stack | --+                            |
       |      |       |   | Statically allocated       |
       |      v       |   | on thread start.           |
       |              | --+                            |
       +--------------+                                |
       |              | -------------------------------+
       +--------------+ brk
       |              | --+
       |      ^       |   | Can grow
```

```
         |       |        |    | dynamically
         |     Heap       | --+
         +--------------+ start_brk
         |     BSS        | --  Static variables initialized to zero
         +--------------+
         |     Data       | --+
         +--------------+    | Binary (disk image)
         |     Code       | --+
         +--------------+
low
addresses
```

Even though this picture might look daunting it is still a simplification. (For a full understanding of the memory subsystem read a book like "Understanding the Linux Kernel" or "Linux System Programming") What I want you to take away from this is that there are two types of dynamically allocatable memory: the heap and memory mapped segments. I will try to call this heap the *C-heap* from now on, to distinguish it from an Erlang process heap. I will call a memory mapped segment for just a *segment*, and any of the stacks in this picture for the *C-stack*.

The C-heap is allocated through malloc and a segment is allocated with mmap.

1. A note on pictures of memory

> **Note** When drawing overview pictures of system memory and stacks we will follow the convention that memory addresses grows upward. That is low memory addresses on the bottom of the page and high memory addresses on the top of the page. (Stacks most often grow downward starting at high addresses, so that new elements are pushed at the lowest address.)
>
> However when we draw a c-structure we will draw the fields from the top and down, even though the first field of the structure will be at the lowest address and the following fields at higher addresses. So pictures of structures have low address at the top of the page and high address at the bottom of the page.
>
> This means that a picture of a c-structure and a picture of a memory area will have their address positions on the page mirrored. This becomes somewhat confusing when we try to pictures structures and heaps in the same picture.

12.1 The memory subsystem

Now that we dive into the memory subsystem it will once again be apparent that ERTS is more like an operating system than just a programming language environment. Not only does ERTS provide a garbage collector for Erlang terms on the Erlang process level, but it also provides a plethora of low level memory allocators and memory allocation strategies.

For an overview of memory allocators see the erts_alloc documentation at: http://www.erlang.org/doc/man/erts_alloc.html

All these allocators also comes with a number of parameters that can be used to tweak their behavior, and this is probably one of the most important areas from an operational point of view. This is where we can configure the system behavior to fit anything from a small embedded control system (like a Raspberry Pi) to an Internet scale 2TB database server.

There are currently eleven different allocators, six different allocation strategies, and more than 18 other different settings, some of which are taking arbitrary numerical values. This means that there basically is an infinite number of possible configurations. (OK, strictly speaking it is not infinite, since each number is bounded, but there are more configurations than you can shake a stick at.)

In order to be able to use these settings in any meaningful way we will have to understand how these allocators work and how each setting impacts the performance of the allocator.

The erts_alloc manual goes as far as to give the following warning:

WARNING

Warning

> Only use these flags if you are absolutely sure what you are doing. Unsuitable settings may cause serious performance degradation and even a system crash at any time during operation.
>
> — Ericsson AB *http://www.erlang.org/doc/man/erts_alloc.html*

Making you absolutely sure that you know what you are doing, that is what this chapter is about.

Oh yes, we will also go into details of how the garbage collector works.

12.2 Different type of memory allocators

The Erlang run-time system is trying its best to handle memory in all situations and under all types of loads, but there are always corner cases. In this chapter we will look at the details of how memory is allocated and how the different allocators work. With this knoweledge and some tools that we will look at later you should be able to detect and fix problems if your system ends up in one of these corner cases.

For a nice story about the troubles the system might get into and how to analyze and correct the behavior read Fred Hébert's essay "Troubleshooting Down the Logplex Rabbit Hole".

When we are talking about a memory allocator in this book we have a specific meaning in mind. Each memory allocator manage allocations and deallocations of memory of a certain type. Each allocator is intended for a specific type of data and is often specialized for one size of data.

Each memory allocator implements the allocator interface but can used different algorithms and settings for the actual memory allocation.

The goal with having different allocators is to reduce fragmentation, by grouping allocations of the same size, and to increase performance, by making frequent allocations cheap.

There are two special, fundamental or generic, memory allocator types *sys_alloc* and *mseg_alloc*, and nine specific allocators implemented through the *alloc_util* framework.

In the following sections we will go though the different allocators, with a little detour into the general framework for allocators (alloc_util).

Each allocator has several names used in the documentation and in the C code. See Table 12.1 for a short list of all allocators and their names. The C-name is used in the C-code to refer to the allocator. The Type-name is used in erl_alloc.types to bind allocation types to an allocator. The Flag is the letter used for setting parameters of that allocator when starting Erlang.

Name	Description	C-name	Type-name	Flag
Basic allocator	malloc interface	sys_alloc	SYSTEM	Y
Memory segment allocator	mmap interface	mseg_alloc	-	M
Temporary allocator	Temporary allocations	temp_alloc	TEMPORARY	T
Heap allocator	Erlang heap data	eheap_alloc	EHEAP	H
Binary allocator	Binary data	binary_alloc	BINARY	B
ETS allocator	ETS data	ets_alloc	ETS	E
Driver allocator	Driver data	driver_alloc	DRIVER	R
Short lived allocator	Short lived memory	sl_alloc	SHORT_LIVED	S
Long lived allocator	Long lived memory	ll_alloc	LONG_LIVED	L
Fixed allocator	Fixed size data	fix_alloc	FIXED_SIZE	F
Standard allocator	For most other data	std_alloc	STANDARD	D

Table 12.1: List of memory allocators.

12.2.1 The basic allocator: sys_alloc

The allocator sys_alloc can not be disabled, and is basically a straight mapping to the underlying OS malloc implementation in libc.

If a specific allocator is disabled then sys_alloc is used instead.

All specific allocators uses either sys_alloc or mseg_alloc to allocate memory from the operating system as needed.

When memory is allocated from the OS sys_alloc can add (pad) a fixed number of kilobytes to the requested number. This can reduce the number system calls by over allocating memory. The default padding is zero.

When memory is freed, sys_alloc will keep some free memory allocated in the process. The size of this free memory is called the trim threshold, and the default is 128 kilobytes. This also reduces the number of system calls at the cost of a higher memory footprint. This means that if you are running the system with the default settings you can experience that the Beam process does not give memory back to the OS directly as memory is freed up.

Memory areas allocated by sys_alloc are stored in the C-heap of the beam process which will grow as needed through system calls to brk.

12.2.2 The memory segment allocator: mseg_alloc

If the underlying operating system supports mmap a specific memory allocator can use mseg_alloc instead of sys_alloc to allocate memory from the operating system.

Memory areas allocated through mseg_alloc are called segments. When a segment is freed it is not immediately returned to the OS, instead it is kept in a segment cache.

When a new segment is allocated a cached segment is reused if possible, i.e. if it is the same size or larger than the requested size but not too large. The value of *absolute max cache bad fit* determines the number of kilobytes of extra size which is considered not too large. The default is 4096 kilobytes.

In order not to reuse a 4096 kilobyte segment for really small allocations there is also a *relative_max_cache_bad_fit* value which states that a cached segment may not be used if it is more than that many percent larger. The default value is 20 percent. That is a 12 KB segment may be used when asked for a 10 KB segment.

The number of entries in the cache defaults to 10 but can be set to any value from zero to thirty.

12.2.3 The memory allocator framework: alloc_util

Building on top of the two generic allocators (sys_alloc and mseg_alloc) is a framework called *alloc_util* which is used to implement specific memory allocators for different types of usage and data.

The framework is implemented in *erl_alloc_util.[ch]* and the different allocators used by ERTS are defined in erl_alloc.types in the directory "erts/emulator/beam/".

In a smp system there is usually one allocator of each type per scheduler thread.

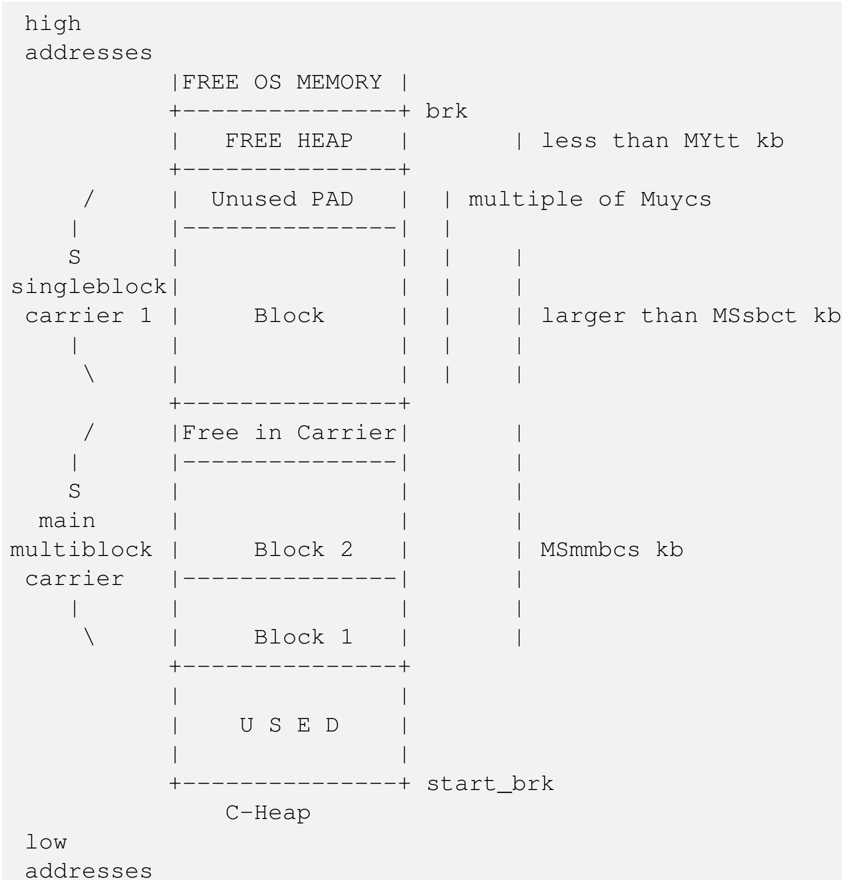
The smallest unit of memory that an allocator work with is called a *block*. When you call an allocator to allocate a certain amount of memory what you get back is a block. It is also blocks that you give as an argument to the allocator when you want to deallocate memory.

The allocator does not allocate blocks from the operating system directly though. Instead the allocator allocates a *carrier* from the operating system, either through sys_alloc or through mseg_alloc, which in turn uses malloc or mmap. If sys_alloc is used the carrier is placed on the C-heap and if mseg_alloc is used the carrier is placed in a segment.

Small blocks are placed in a multiblock carrier. A multiblock carrier can as the name suggests contain many blocks. Larger blocks are placed in a singleblock carrier, which as the name implies on contains one block.

What's considered a small and a large block is determined by the parameter *singleblock carrier threshold* (`sbct`), see the list of system flags below.

Most allocators also have one "main multiblock carrier" which is never deallocated.



```
high
addresses
          |FREE OS MEMORY |
          +---------------+ brk
          |   FREE HEAP   |       | less than MYtt kb
          +---------------+
     /    |  Unused PAD   |  | multiple of Muycs
    |     |---------------|  |
    S     |               |  |    |
singleblock|              |  |    |
 carrier 1 |     Block    |  |    | larger than MSsbct kb
    |     |               |  |    |
     \    |               |  |    |
          +---------------+
     /    |Free in Carrier|       |
    |     |---------------|       |
    S     |               |       |
  main    |               |       |
multiblock |     Block 2  |       | MSmmbcs kb
 carrier  |---------------|       |
    |     |               |       |
     \    |     Block 1   |       |
          +---------------+
          |               |
          |    U S E D    |
          |               |
          +---------------+ start_brk
              C-Heap
 low
 addresses
```

12.2.3.1 Memory allocation strategies

To find a free block of memory in a multi block carrier an allocation strategy is used. Each type of allocator has a default allocation strategy, but you can also set the allocation strategy with the `as` flag.

The Erlang Run-Time System Application Reference Manual lists the following allocation strategies:

> *Best fit*: Find the smallest block that satisfies the requested block size. (bf)
>
> *Address order best fit*: Find the smallest block that satisfies the requested block size. If multiple blocks are found, choose the one with the lowest address. (aobf)
>
> *Address order first fit*: Find the block with the lowest address that satisfies the requested block size. (aoff)
>
> *Address order first fit carrier best fit* Find the carrier with the lowest address that can satisfy the requested block size, then find a block within that carrier using the "best fit" strategy. (aoffcbf)
>
> *Address order first fit carrier address order best fit*: Find the carrier with the lowest address that can satisfy the requested block size, then find a block within that carrier using the "address order best fit" strategy. aoffcaobf (address order first fit carrier address order best fit)
>
> *Good fit*: Try to find the best fit, but settle for the best fit found during a limited search. (gf)
>
> *A fit*: Do not search for a fit, inspect only one free block to see if it satisfies the request. This strategy is only intended to be used for temporary allocations. (af)
>
> — erts_alloc

12.2.4 The temporary allocator: temp_alloc

The allocator *temp_alloc*, is used for temporary allocations. That is very short lived allocations. Memory allocated by temp_alloc may not be allocated over a Erlang process context switch.

You can use temp_alloc as a small scratch or working area while doing some work within a function. Look at it as an extension of the C-stack and free it in the same way. That is, to be on the safe side, free memory allocated by temp_alloc before returning from the function that did the allocation. There is a note in erl_alloc.types saying that you should free a temp_alloc block before the emulator starts executing Erlang code.

Note that no Erlang process running on the same scheduler as the allocator may start executing Erlang code before the block is freed. This means that you can not use a temporary allocation over a bif or nif trap (yield).

In a default R16 smp system there is N+1 temp_alloc allocators where N is the number of schedulers. The temp_alloc uses the "A fit" (`af`) strategy. Since the allocation pattern of the temp_alloc basically is that of a stack (mostly of size 0 or 1), this strategy works fine.

The temporary allocator is, in R16, used by the following types of data: TMP_HEAP, MSG_ROOTS, ROOTSET, LOADER_TEMP, NC_TMP, TMP, DCTRL_BUF, TMP_DIST_BUF, ESTACK, DB_TMP, DB_MC_STK, DB_MS_CMPL_HEAP, LOGGER_DSBUF, TMP_DSBUF, DDLL_TMP_BUF, TEMP_TERM, SYS_READ_BUF, ENVIRONMENT, CON_VPRINT_BUF.

For an up to date list of allocation types allocated with each allocator, see erl_alloc.types (e.g. `grep TEMPORARY erts/emulator/beam/erl_alloc.types`).

I will not go through each of these different types, but in general as you can guess by their names, they are temporary buffers or work stacks.

12.2.5 The heap allocator: eheap_alloc

The heap allocator, is used for allocating memory blocks where tagged Erlang terms are stored, such as Erlang process heaps (all generations), heap fragments, and the beam_registers.

This is probably the memory areas you are most interested in as an Erlang developer or when tuning an Erlang system. We will talk more about how these areas are managed in the upcoming sections on garbage collection and process memory. There we will also cover what a heap fragment is.

12.2.6 The binary allocator: binary_alloc

The binary allocator is used for, yes you guessed it, binaries. Binaries can be of quite varying sizes and have varying life spans. This allocator uses the *best fit* allocation strategy by default.

12.2.7 The ETS allocator: ets_alloc

The ETS allocator is used for most ets related data, except for some short lived or temporary data used by ets tables-

12.2.8 The driver allocator: driver_alloc

The driver allocator is used for ports, linked in drivers and nifs.

12.2.9 The short lived allocator: sl_alloc

The short lived allocator is used for lists and buffers that are expected to be short lived. Short lived data can live longer than temporary data.

12.2.10 The long lived allocator: ll_alloc

The long lived allocator is used for long lived data, such as atoms, modules, funs and long lived tables

12.2.11 The fixed size allocator: fix_alloc

The fixed allocator is used for objects of a fixed size, such as PCBs, message refs and a few other. The fixed size allocator uses the *address order best fit* allocation strategy by default.

12.2.12 The standard allocator: std_alloc

The standard allocator is used by the other types of data. (active_procs alloc_info_request arg_reg bif_timer_ll bits_buf bpd calls_buf db_heir_data db_heir_data db_named_table_entry dcache ddll_handle ddll_processes ddll_processes dist_entry dist_tab driver_lock ethread_standard fd_entry_buf fun_tab gc_info_request io_queue line_buf link_lh module_refs monitor_lh monitor_lh monitor_sh nlink_lh nlink_lh nlink_sh node_entry node_tab nodes_monitor port_data_heap port_lock port_report_exit port_specific_data proc_dict process_specific_data ptimer_ll re_heap reg_proc reg_tab sched_wall_time_request stack suspend_monitor thr_q_element thr_queue zlib)

12.3 TODO: system flags for memory

TODO

12.4 Process Memory

As we saw in Chapter 3 a process i really just a number of memory areas, in this chapter we will look a bit closer at how the stack, the heap and the mailbox are managed.

The default size of the stack and heap is 233 words. This default size can be changed globally when starting Erlang through the `+h + flag.You can also set the minimum heap size by when starting a process with +spawn_opt` by setting `min_heap_size`.

Erlang terms are tagged as we saw in Chapter 4, and when they are stored on the heap they are either cons cells or boxed objects.

12.4.1 Term sharing

Objects on the heap are passed by references within the context of one process. If you call one function with a tuple as an argument, then only a tagged reference to that tuple is passed to the called function. When you build new terms you will also only use references to sub terms.

For example if you have the string "hello" (which is the same as this list of integers: [104,101,108,108,111]) you would get a stack layout similar to:

```
        ADR                                BINARY  VALUE  +  DESCRIPTION
hend ->     +-------- -------- -------- --------+
            |              ...                  |
            |              ...                  |
            |00000000 00000000 00000000 10000001| 128 + list tag  ---------------+
stop ->;     |                                   |                               |
                                                                                |
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 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 ->     +------------------------------------+
```

If you then create a tuple with two instances of the list, all that is repeated is the tagged pointer to the list: 0000000000000000000000001
The code

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

would result in a memory layout as seen below. That is, a boxed header saying that this is a tuple of size 2 and then two pointers to the same list.

```
ADR VALUE                            DESCRIPTION
144 00000000000000000000000001000001 128+CONS
140 00000000000000000000000001000001 128+CONS
136 00000000000000000000000010000000 2+ARITYVAL
```

This is nice, since it is cheap to do and uses very little space. But if you send the tuple to another process or do any other type of IO, or any operations which results in something called a *deep copy*, then the data structure is expanded. So if we send out tuple `T` to another process P2 (`P2 ! T`) then the heap of T2 will look like in:

```
 ..
```

You can quickly bring down your Erlang node by expanding a highly shared term, see share.erl.

```
-module(share).

-export([share/2, size/0]).

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

size() ->
    T = share:share(5,[a,b,c]),
    {{size, erts_debug:size(T)},
     {flat_size, erts_debug:flat_size(T)}}.



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

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

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

You can calculate the memory size of a shared term and the size of the expanded size of the term with the functions `erts_debug:size/1` and `erts_debug:flat_size/1`.

```
> share:size().
{{size,19386},{flat_size,94110}}
```

For most applications this is not a problem, but you should be aware of the problem, which can come up in many situations. A deep copy is used for IO, ETS tables, binary_to_term, and message passing.

Let us look in more detail how message passing works.

12.4.2 Message passing

When a process P1 sends a message M to another (local) process P2, the process P1 first calculates the flat size of M. Then it allocates a new message buffer of that size by doing a heap_alloc of a heap_frag in the local scheduler context.

Given the code in send.erl the state of the system could look like this just before the send in p1/1:

```
    x0       |00000000 00000000 00000000 00100011| Pid 2
    x1       |00000000 00000000 00000000 01001010| 136 + boxed tag -----------+
                                                                               |
                                                                               |
         ADR                              BINARY  VALUE  +  DESCRIPTION       |
 hend ->     +-------- -------- -------- --------+                            |
             |              ...                  |                            |
             |              ...                  |                            |
 stop ->     |                                   |                            |
                                                                               |
 htop ->     |                                   |                            |
         144 |00000000 00000000 00000000 01000001| 128+CONS        ---------------+
         140 |00000000 00000000 00000000 01000001| 128+CONS        ---------------+
         136 |00000000 00000000 00000000 10000000| 2+ARITYVAL             <---+   |
         132 |00000000 00000000 00000000 01111001| 120+CONS        -------------- | -+
         128 |00000000 00000000 00000110 10001111| (H) 104 bsl 4 + small int tag <+  |
         124 |00000000 00000000 00000000 01110001| 112+CONS        ----------------- | -+
         120 |00000000 00000000 00000110 01011111| (e) 101 bsl 4 + small int tag <---+  |
         116 |00000000 00000000 00000000 01110001| 112+CONS        -------------------- |  ↩
            -+
         112 |00000000 00000000 00000110 11001111| (l) 108 bsl 4 + small int tag <------+   ↩
             |
         108 |00000000 00000000 00000000 01110001|  96+CONS        -----------------------   ↩
             | -+
         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 ->     +-----------------------------------+


P2
```

When P1 start sending the message M to P2. It (through the code in erl_message.c) first calculates the flat size of M (which in our example is 23 words)[1]. Then (in a SMP system) if it can take a lock on P2 and there is enough room on the heap of P2 it will copy the message to the heap of P2.

If P2 is running (or exiting) or there isn't enough space on the heap, then a new heap fragment is allocated (of sizeof ErlHeapFragment - sizeof(Eterm) + 23*sizeof(Eterm)) [2] which after initialization will look like:

```
erl_heap_fragment:
    ErlHeapFragment* next;          NULL
    ErlOffHeap off_heap:
      erl_off_heap_header* first; NULL
```

[1] We ignore tracing here which will add a trace token to the size of the message, and always use a heap fragment.
[2] The -sizeof(Eterm) comes from mem in ErlHeapFragment already having the size of 1 Eterm

```
      Uint64 overhead;              0
    unsigned alloc_size;           23
    unsigned used_size;            23
    Eterm mem[1];                   ?
      ... 22 free words
```

Then the message is copied into the heap fragment:

```
erl_heap_fragment:
    ErlHeapFragment* next;         NULL
    ErlOffHeap off_heap:
      erl_off_heap_header* first; Boxed tag+&amp;mem+2*WS-+
      Uint64 overhead;               0                 |
    unsigned alloc_size;            23                 |
    unsigned used_size;             23                 |
    Eterm mem:                    2+ARITYVAL   <------+
                                  &amp;mem+3*WS+1  ---+
                                  &amp;mem+13*WS+1 ------+
                                  (H*16)+15    <--+  |
                                  &amp;mem+5*WS+1  --+    |
                                  (e*16)+15    <-+    |
                                  &amp;mem+7*WS+1  ----| |
                                  (l*16)+15    <---+ |
                                  &amp;mem+9*WS+1  ---+  |
                                  (l*16)+15    <--+  |
                                  &amp;mem+11*WS+1 ----+ |
                                  (o*16)+15    <---+ |
                                  NIL                |
                                  (H*16)+15    <-----+
                                  &amp;mem+15*WS+1 --+
                                  (e*16)+15    <-+
                                  &amp;mem+17*WS+1 ----|
                                  (l*16)+15    <---+
                                  &amp;mem+19*WS+1 ---+
                                  (l*16)+15    <--+
                                  &amp;mem+21*WS+1 ----+
                                  (o*16)+15    <---+
                                  NIL</pre>
```

In either case a a new mbox (`ErlMessage`) is allocated, a lock (`ERTS_PROC_LOCK_MSGQ`) is taken on the receiver and the message on the heap or the in the new heap fragment is linked into the mbox.

```
 erl_mesg {
    struct erl_mesg* next = NULL;
    data:  ErlHeapFragment *heap_frag = bp;
    Eterm m[0]              = message;
 } ErlMessage;
```

Then the mbox is linked into the in message queue (`msg_inq`) of the receiver, and the lock is released. Note that `msg_inq.last` points to the `next` field of the last message in the queue. When a new mbox is linked in this next pointer is updated to point to the new mbox, and the last pointer is updated to point to the next field of the new mbox.

12.4.3 Binaries

As we saw in Chapter 4 there are four types of binaries internally. Three of these types, *heap binaries*, *sub binaries* and *match contexts* are stored on the local heap and handled by the garbage collector and message passing as any other object, copied as needed.

12.4.3.1 Reference Counting

The fourth type. large binaries or *refc binaries* on the other hand are partially stored outside of the process heap and they are reference counted.

The payload of a refc binary is stored in memory allocated by the binary allocator. There is also a small reference to the payload call a ProcBin which is stored on the process heap. This reference is copied by message passing and by the GC, but the payload is untouched. This makes it relatively cheap to send large binaries to other processes since the whole binary doesn't need to be copied.

All references through a ProcBin to a refc binary increases the reference count of the binary by one. All ProcBin objects on a process heap are linked together in a linked list. After a GC pass this linked list is traversed and the reference count of the binary is decreased with one for each ProcBin that has deceased. If the reference count of the refc binary reaches zero that binary is deallocated.

Having large binaries reference counted and not copied by send or garbage collection is a big win, but there is one problem with having a mixed environment of garbage collection and reference counting. In a pure reference counted implementation the reference count would be reduce as soon as a reference to the object dies, and when the reference count reaches zero the object is freed. In the ERTS mixed environment a reference to a reference counted object does not die until a garbage collection detects that the reference is dead.

This means that binaries, which has a tendency to be large or even huge, can hang around for a long time after all references to the binary are dead. Note that since binaries are allocated globally, all references from all processes need to be dead, that is all processes that has seen a binary need to do a GC.

Unfortunately it is not always easy, as a developer, to see which processes have seen a binary in the GC sense of the word seen. Imagine for example that you have a load balancer that receives work items and dispatches them to workers.

In this code there is an example of a loop which doesn't need to do GC. (See listing lb for a full example.)

```
loop(Workers, N) ->
  receive
    WorkItem ->
       Worker = lists:nth(N+1, Workers),
       Worker ! WorkItem,
       loop(Workers, (N+1) rem length(Workers))
  end.
```

This server will just keep on grabbing references to binaries and never free them, eventually using up all system memory.

When one is aware of the problem it is easy to fix, one can either do a garbage_collect on each iteration of *loop* or one could do it every five seconds or so by adding an after clause to the receive. (*after 5000 → garbage_collect(), loop(Workers, N)*).

12.4.3.2 Sub Binaries and Matching

When you match out a part of a binary you get a sub binary. This sub binary will be a small structure just containing pointers into the real binary. This increases the reference count for the binary but uses very little extra space.

If a match would create a new copy of the matched part of the binary it would cost both space and time. So in most cases just doing a pattern match on a binary and getting a sub binary to work on is just what you want.

There are some degenerate cases, imagine for example that you load huge file like a book into memory and then you match out a small part like a chapter to work on. The problem is then that the whole of the rest of the book is still kept in memory until you are done with processing the chapter. If you do this for many books, perhaps you want to get the introduction of every book in your file system, then you will keep the whole of each book in memory and not just the introductory chapter. This might lead to huge memory usage.

The solution in this case, when you know you only want one small part of a large binary and you want to have the small part hanging around for some time, is to use `binary:copy/1`. This function is only used for its side effect, which is to actually copy the sub binary out of the real binary removing the reference to the larger binary and therefore hopefully letting it be garbage collected.

There is a pretty thorough explanation of how binary construction and matching is done in the Erlang documentation: http://www.erlang.org/doc/efficiency_guide/binaryhandling.html.

12.4.4 Garbage Collection

When a process runs out of space on the stack and heap the process will try to reclaim space by doing a minor garbage collection. The code for this can be found in erl_gc.c.

ERTS uses a generational copying garbage collector. A copying collector means that during garbage collection all live young terms are copied from the old heap to a new heap. Then the old heap is discarded. A generational collector works on the principle that most terms die young, they are temporary terms created, used, and thrown away. Older terms are promoted to the old generation which is collected more seldom, with the rational that once a term has become old it will probably live for a long time.

Conceptually a garbage collection cycle works as follows:

- First you collect all roots (e.g. the stack).
- Then for each root, if the root points to a heap allocated object which doesn't have a forwarding pointer you copy the object to the new heap. For each copied object update the original with a forwarding pointer to the new copy.
- Now go through the new heap and do the same as for the roots.

We will go through an example to see how this is done in detail. We will go through a minor collection without an old generation, and we will only use the stack as the root set. In reality the process dictionary, trace data and probe data among other things are also included in the rootset.

Let us look at how the call to garbage_collect in the gc_example behaves. The code will generate a string which is shared by two elements of a cons and a tuple, the tuple will the be eliminated resulting in garbage. After the GC there should only be one string on the heap. That is, first we generate the term `{["Hello","Hello"], "Hello"}` (sharing the same string "Hello" in all instances. Then we just keep the term `["Hello","Hello"]` when triggering a gc.

Note
We will take the opportunity to go through how you, on a linux system, can used gdb to examine the behavior of ERTS. You can of course use the debugger of your choice. If you already know how to use gdb or if you have no interest in going into the debugger you can just ignore the meta text about how to inspect the system and just look at the diagrams and the explanations of how the GC works.

```
-module(gc_example).
-export([example/0]).

example() ->
  T = gen_data(),
  S = element(1, T),
  erlang:garbage_collect(),
  S.

gen_data() ->
 S = gen_string($H, $e, $l, $l, $o),
 T = gen_tuple([S,S],S),
 T.

gen_string(A,B,C,D,E) ->
   [A,B,C,D,E].

gen_tuple(A,B) ->
 {A,B}.
```

After compiling the example I start an erlang shell, test the call and prepare for a new call to the example (without hitting return):

```
1> gc_example:example().
["Hello","Hello"]
2> spawn(gc_example,example,[]).
```

Then I use gdb to attach to my erlang node (os PID: 2955 in this case)

```
$ gdb /home/happi/otp/lib/erlang/erts-6.0/bin/beam.smp 2955
```

Note

Depending on your settings for ptrace_scope you might have to precede the gdb invocation with *sudo*.

Then in gdb I set a breakpoint at the start of the main GC function and let the node continue:

```
(gdb) break garbage_collect_0
(gdb) cont
Continuing.
```

Now I hit enter in the Erlang shell and execution stops at the breakpoint:

```
Breakpoint 1, garbage_collect_0 (A__p=0x7f673d085f88, BIF__ARGS=0x7f673da90340) at beam/bif ↩
    .c:3771
3771        FLAGS(BIF_P) |= F_NEED_FULLSWEEP;
```

Now we can inspect the PCB of the process:

```
(gdb) p *(Process *) A__p
$1 = {common = {id = 1408749273747, refc = {counter = 1}, tracer_proc =  ↩
    18446744073709551611, trace_flags = 0, u = {alive = {
        started_interval = 0, reg = 0x0, links = 0x0, monitors = 0x0, ptimer = 0x0},  ↩
            release = {later = 0, func = 0x0, data = 0x0,
        next = 0x0}}}, htop = 0x7f6737145950, stop = 0x7f6737146000, heap = 0x7f67371458c8, ↩
            hend = 0x7f6737146010, heap_sz = 233,
  min_heap_size = 233, min_vheap_size = 46422, fp_exception = 0, hipe = {nsp = 0x0, nstack  ↩
      = 0x0, nstend = 0x0, ncallee = 0x7f673d080000,
    closure = 0, nstgraylim = 0x0, nstblacklim = 0x0, ngra = 0x0, ncsp = 0x7f673d0863e8,  ↩
        narity = 0, float_result = 0}, arity = 0,
  arg_reg = 0x7f673d086080, max_arg_reg = 6, def_arg_reg = {393227, 457419,  ↩
      18446744073709551611, 233, 46422, 2000}, cp = 0x7f673686ac40,
  i = 0x7f673be17748, catches = 0, fcalls = 1994, rcount = 0, schedule_count = 0, reds = 0, ↩
       group_leader = 893353197987, flags = 0,
  fvalue = 18446744073709551611, freason = 0, ftrace = 18446744073709551611, next = 0 ↩
      x7f673d084cc0, nodes_monitors = 0x0,
  suspend_monitors = 0x0, msg = {first = 0x0, last = 0x7f673d086120, save = 0x7f673d086120, ↩
       len = 0, mark = 0x0, saved_last = 0x7d0}, u = {
    bif_timers = 0x0, terminate = 0x0}, dictionary = 0x0, seq_trace_clock = 0,  ↩
        seq_trace_lastcnt = 0,
  seq_trace_token = 18446744073709551611, initial = {393227, 457419, 0}, current = 0 ↩
      x7f673be17730, parent = 1133871366675,
  approx_started = 1407857804, high_water = 0x7f67371458c8, old_hend = 0x0, old_htop = 0x0, ↩
       old_heap = 0x0, gen_gcs = 0,
  max_gen_gcs = 65535, off_heap = {first = 0x0, overhead = 0}, mbuf = 0x0, mbuf_sz = 0, psd ↩
       = 0x0, bin_vheap_sz = 46422,
  bin_vheap_mature = 0, bin_old_vheap_sz = 46422, bin_old_vheap = 0, sys_task_qs = 0x0,  ↩
      state = {counter = 41002}, msg_inq = {first = 0x0,
    last = 0x7f673d086228, len = 0}, pending_exit = {reason = 0, bp = 0x0}, lock = {flags = ↩
         {counter = 1}, queue = {0x0, 0x0, 0x0, 0x0},
    refc = {counter = 1}}, scheduler_data = 0x7f673bd6c080, suspendee =  ↩
        18446744073709551611, pending_suspenders = 0x0, run_queue = {
    counter = 140081362118912}, hipe_smp = {have_receive_locks = 0}}
```

Wow, that was a lot of information. The interesting part is about the stack and the heap:

```
hend = 0x7f6737146010,
stop = 0x7f6737146000,
htop = 0x7f6737145950,
heap = 0x7f67371458c8,
```

By using some helper scripts we can inspect the stack and the heap in a meaningful way. (see Appendix .14 for the definitions of the scripts in gdb_script.)

```
(gdb) source gdb_scripts
(gdb) print_p_stack A__p
0x00007f6737146008 [0x00007f6737145929] cons -> 0x00007f6737145928
(gdb) print_p_heap A__p
0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f6737145919] cons -> 0x00007f6737145918
0x00007f6737145928 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145920 [0xfffffffffffffffb] NIL
0x00007f6737145918 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145910 [0x00007f67371458f9] cons -> 0x00007f67371458f8
0x00007f6737145908 [0x000000000000048f] 72
0x00007f6737145900 [0x00007f67371458e9] cons -> 0x00007f67371458e8
0x00007f67371458f8 [0x000000000000065f] 101
0x00007f67371458f0 [0x00007f67371458d9] cons -> 0x00007f67371458d8
0x00007f67371458e8 [0x00000000000006cf] 108
0x00007f67371458e0 [0x00007f67371458c9] cons -> 0x00007f67371458c8
0x00007f67371458d8 [0x00000000000006cf] 108
0x00007f67371458d0 [0xfffffffffffffffb] NIL
0x00007f67371458c8 [0x00000000000006ff] 111
```

Here we can see the heap of the process after it has allocated the list "Hello" on the heap and the cons containing that list twice, and the tuple containing the cons and the list. The *root set*, in this case the stack, contains a pointer to the cons containing two copies of the list. The tuple is dead, that is, there are no references to it.

The garbage collection starts by calculating the root set and by allocating a new heap (*to space*). By stepping into the gc code in the debugger you can see how this is done. I will not go through the details here. After a number of steps the execution will reach the point where all terms in the root set are copied to the new heap. This starts around (depending on version) line 1272 with a `while` loop in erl_gc.c.

In our case the root is a cons pointing to address 0x00007f95666597f0 containing the letter (integer) *H*. When a cons cell is moved from the current heap, called *from space*, to *to space* the value in the head (or car) is overwritten with a *moved cons* tag (the value 0).

After the first step where the root set is moved, the from space and the to space looks like this:

from space:

```
(gdb) print_p_heap p
0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f67371445b1] cons -> 0x00007f67371445b0
0x00007f6737145928 [0x0000000000000000] Tuple size 0
0x00007f6737145920 [0xfffffffffffffffb] NIL
0x00007f6737145918 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145910 [0x00007f67371458f9] cons -> 0x00007f67371458f8
0x00007f6737145908 [0x000000000000048f] 72
0x00007f6737145900 [0x00007f67371458e9] cons -> 0x00007f67371458e8
0x00007f67371458f8 [0x000000000000065f] 101
0x00007f67371458f0 [0x00007f67371458d9] cons -> 0x00007f67371458d8
0x00007f67371458e8 [0x00000000000006cf] 108
0x00007f67371458e0 [0x00007f67371458c9] cons -> 0x00007f67371458c8
```

```
0x00007f67371458d8 [0x00000000000006cf] 108
0x00007f67371458d0 [0xfffffffffffffffb] NIL
0x00007f67371458c8 [0x00000000000006ff] 111
```

to space:

```
(gdb) print_heap n_htop-1 n_htop-2
0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f6737145918
0x00007f67371445b0 [0x00007f6737145909] cons -> 0x00007f6737145908
```

In from space the head of the first cons cell has been overwritten with 0 (looks like a tuple of size 0) and the tail has been overwritten with a forwarding pointer pointing to the new cons cell in the to space. In to space we now have the first cons cell with two backward pointers to the head and the tail of the cons in the from space.

When the collector is done with the root set the to space contains backward pointers to all still live terms. At this point the collector starts sweeping the to space. It uses two pointers `n_hp` pointing to the bottom of the unseen heap and `n_htop` pointing to the top of the heap.

```
n_htop:
        0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f6737145918
n_hp    0x00007f67371445b0 [0x00007f6737145909] cons -> 0x00007f6737145908
```

The GC will then look at the value pointed to by `n_hp`, in this case a cons pointing back to the from space. So it moves that cons to the to space, incrementing n_htop to make room for the new cons, and incrementing `n_hp` to indicate that the first cons is seen.

```
from space:

0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f67371445b1] cons -> 0x00007f67371445b0
0x00007f6737145928 [0x0000000000000000] Tuple size 0
0x00007f6737145920 [0xfffffffffffffffb] NIL
0x00007f6737145918 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145910 [0x00007f67371445c1] cons -> 0x00007f67371445c0
0x00007f6737145908 [0x0000000000000000] Tuple size 0
0x00007f6737145900 [0x00007f67371458e9] cons -> 0x00007f67371458e8
0x00007f67371458f8 [0x000000000000065f] 101
0x00007f67371458f0 [0x00007f67371458d9] cons -> 0x00007f67371458d8
0x00007f67371458e8 [0x00000000000006cf] 108
0x00007f67371458e0 [0x00007f67371458c9] cons -> 0x00007f67371458c8
0x00007f67371458d8 [0x00000000000006cf] 108
0x00007f67371458d0 [0xfffffffffffffffb] NIL
0x00007f67371458c8 [0x00000000000006ff] 111

to space:

n_htop:
        0x00007f67371445c8 [0x00007f67371458f9] cons -> 0x00007f67371458f8
        0x00007f67371445c0 [0x000000000000048f] 72
n_hp    0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f6737145918
SEEN    0x00007f67371445b0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
```

The same thing then happens with the second cons.

```
from space:

0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f67371445b1] cons -> 0x00007f67371445b0
```

```
0x00007f6737145928 [0x0000000000000000] Tuple size 0
0x00007f6737145920 [0x00007f67371445d1] cons -> 0x00007f67371445d0
0x00007f6737145918 [0x0000000000000000] Tuple size 0
0x00007f6737145910 [0x00007f67371445c1] cons -> 0x00007f67371445c0
0x00007f6737145908 [0x0000000000000000] Tuple size 0
0x00007f6737145900 [0x00007f67371458e9] cons -> 0x00007f67371458e8
0x00007f67371458f8 [0x000000000000065f] 101
0x00007f67371458f0 [0x00007f67371458d9] cons -> 0x00007f67371458d8
0x00007f67371458e8 [0x00000000000006cf] 108
0x00007f67371458e0 [0x00007f67371458c9] cons -> 0x00007f67371458c8
0x00007f67371458d8 [0x00000000000006cf] 108
0x00007f67371458d0 [0xfffffffffffffffb] NIL
0x00007f67371458c8 [0x00000000000006ff] 111

to space:

n_htop:
        0x00007f67371445d8 [0xfffffffffffffffb] NIL
        0x00007f67371445d0 [0x00007f6737145909] cons -> 0x00007f6737145908
        0x00007f67371445c8 [0x00007f67371458f9] cons -> 0x00007f67371458f8
n_hp    0x00007f67371445c0 [0x000000000000048f] 72
SEEN    0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f67371445d0
SEEN    0x00007f67371445b0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
```

The next element in to space is the immediate 72, which is only stepped over (with `n_hp++`). Then there is another cons which is moved.

The same thing then happens with the second cons.

```
from space:

0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f67371445b1] cons -> 0x00007f67371445b0
0x00007f6737145928 [0x0000000000000000] Tuple size 0
0x00007f6737145920 [0x00007f67371445d1] cons -> 0x00007f67371445d0
0x00007f6737145918 [0x0000000000000000] Tuple size 0
0x00007f6737145910 [0x00007f67371445c1] cons -> 0x00007f67371445c0
0x00007f6737145908 [0x0000000000000000] Tuple size 0
0x00007f6737145900 [0x00007f67371445e1] cons -> 0x00007f67371445e0
0x00007f67371458f8 [0x0000000000000000] Tuple size 0
0x00007f67371458f0 [0x00007f67371458d9] cons -> 0x00007f67371458d8
0x00007f67371458e8 [0x00000000000006cf] 108
0x00007f67371458e0 [0x00007f67371458c9] cons -> 0x00007f67371458c8
0x00007f67371458d8 [0x00000000000006cf] 108
0x00007f67371458d0 [0xfffffffffffffffb] NIL
0x00007f67371458c8 [0x00000000000006ff] 111

to space:

n_htop:
        0x00007f67371445e8 [0x00007f67371458e9] cons -> 0x00007f67371458e8
        0x00007f67371445e0 [0x000000000000065f] 101
        0x00007f67371445d8 [0xfffffffffffffffb] NIL
n_hp    0x00007f67371445d0 [0x00007f6737145909] cons -> 0x00007f6737145908
SEEN    0x00007f67371445c8 [0x00007f67371458f9] cons -> 0x00007f67371445e0
SEEN    0x00007f67371445c0 [0x000000000000048f] 72
SEEN    0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f67371445d0
SEEN    0x00007f67371445b0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
```

Now we come to a cons that points to a cell that has already been moved. The GC sees the IS_MOVED_CONS tag at

0x00007f6737145908 and copies the destination of the moved cell from the tail (`*n_hp+ = ptr[1];+`). This way sharing is preserved during GC. This step does not affect from space, but the backward pointer in to space is rewritten.

```
to space:

n_htop:
        0x00007f67371445e8 [0x00007f67371458e9] cons -> 0x00007f67371458e8
        0x00007f67371445e0 [0x000000000000065f] 101
n_hp    0x00007f67371445d8 [0xfffffffffffffffb] NIL
SEEN    0x00007f67371445d0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
SEEN    0x00007f67371445c8 [0x00007f67371458f9] cons -> 0x00007f67371445e0
SEEN    0x00007f67371445c0 [0x000000000000048f] 72
SEEN    0x00007f67371445b8 [0x00007f6737145919] cons -> 0x00007f67371445d0
SEEN    0x00007f67371445b0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
```

Then the rest of the list (the string) is moved.

```
from space:

0x00007f6737145948 [0x00007f6737145909] cons -> 0x00007f6737145908
0x00007f6737145940 [0x00007f6737145929] cons -> 0x00007f6737145928
0x00007f6737145938 [0x0000000000000080] Tuple size 2
0x00007f6737145930 [0x00007f67371445b1] cons -> 0x00007f67371445b0
0x00007f6737145928 [0x0000000000000000] Tuple size 0
0x00007f6737145920 [0x00007f67371445d1] cons -> 0x00007f67371445d0
0x00007f6737145918 [0x0000000000000000] Tuple size 0
0x00007f6737145910 [0x00007f67371445c1] cons -> 0x00007f67371445c0
0x00007f6737145908 [0x0000000000000000] Tuple size 0
0x00007f6737145900 [0x00007f67371445e1] cons -> 0x00007f67371445e0
0x00007f67371458f8 [0x0000000000000000] Tuple size 0
0x00007f67371458f0 [0x00007f67371445f1] cons -> 0x00007f67371445f0
0x00007f67371458e8 [0x0000000000000000] Tuple size 0
0x00007f67371458e0 [0x00007f6737144601] cons -> 0x00007f6737144600
0x00007f67371458d8 [0x0000000000000000] Tuple size 0
0x00007f67371458d0 [0x00007f6737144611] cons -> 0x00007f6737144610
0x00007f67371458c8 [0x0000000000000000] Tuple size 0

to space:

n_htop:
n_hp
SEEN    0x00007f6737144618 [0xfffffffffffffffb] NIL
SEEN    0x00007f6737144610 [0x00000000000006ff] 111
SEEN    0x00007f6737144608 [0x00007f6737144611] cons -> 0x00007f6737144610
SEEN    0x00007f6737144600 [0x00000000000006cf] 108
SEEN    0x00007f67371445f8 [0x00007f6737144601] cons -> 0x00007f6737144600
SEEN    0x00007f67371445f0 [0x00000000000006cf] 108
SEEN    0x00007f67371445e8 [0x00007f67371445f1] cons -> 0x00007f67371445f0
SEEN    0x00007f67371445e0 [0x000000000000065f] 101
SEEN    0x00007f67371445d8 [0xfffffffffffffffb] NIL
SEEN    0x00007f67371445d0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
SEEN    0x00007f67371445c8 [0x00007f67371445e1] cons -> 0x00007f67371445e0
SEEN    0x00007f67371445c0 [0x000000000000048f] 72
SEEN    0x00007f67371445b8 [0x00007f67371445d1] cons -> 0x00007f67371445d0
SEEN    0x00007f67371445b0 [0x00007f67371445c1] cons -> 0x00007f67371445c0
```

There are some things to note from this example. When terms are created in Erlang they are created bottom up, starting with the elements. The garbage collector works top down, starting with the top level structure and then copying the elements. This means that the direction of the pointers change after the first GC. This has no real implications but it is good to know when looking at actual heaps. You can not assume that structures should be bottom up.

Also note that the GC does a breath first traversal. This means that locality for one term most often is worse after a GC. With

the size of modern cashes this should not be a problem. You could of course create a pathological example where it becomes a problem, but you can also create a pathological example where a depth first approach would cause problems.

The third thing to note is that sharing is preserved which is really important otherwise we might end up using more space after a GC than before.

Generations. . .

```
hend ->  +----+
         |....|
stop ->  |    |
         |    |    +----+ old_hend
         |    |    |    |
htop ->  |    |    |    |
         |....|    |    | old_htop
         |....|    |....|
heap ->  +----+    +----+ old_heap
        The Heap   Old Heap
```

`+high_water, old_hend, old_htop, old_heap,`
`gen_gcs, max_gen_gcs, off_heap,  mbuf, mbuf_sz, psd, bin_vheap_sz,`
`bin_vheap_mature, bin_old_vheap_sz, bin_old_vheap+.`

12.5 Other interesting memory areas

12.5.1 The atom table.

TODO ==== Code TODO ==== Constants TODO

Chapter 13

Advanced data structures (ETS, DETS, Mnesia)

Chapter 14

IO, Ports and Networking (10p)

Within Erlang all communication is done by asynchronous signaling. The communication between an Erlang node and the outside world is done through a *port*. A port is an interface between Erlang processes and an external resource. In early versions of Erlang a port behaved very much in the same way as a process and you comunicated by sending and receiving signals. You can still communicate with ports this way but there are also a number of BIFs to communicate directly with a port.

In this chapter we will look at how ports are used as a common interface for all IO, how ports comunicate with the outside world and how Erlang processes comunicate with ports. But first we will look at how standard IO works on a higher level.

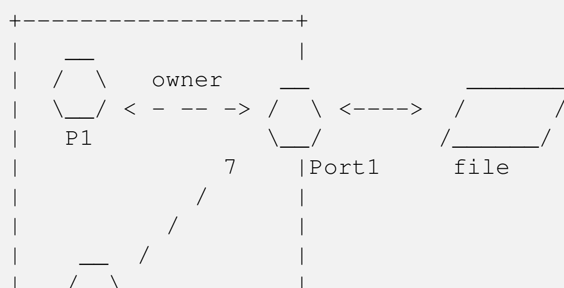
14.1 Standard IO

- IO protocol
- group leader
- erlang:display — a biff that sends directly to node's std out
- io:format — sends through io protocol and the group leader
- Redirecting standard IO at startup (detached mode)
- Standard in and out

14.2 Ports

A port is the process like interface between Erlang processes and everything that is not Erlang processes. The programmer can to a large extent pretend that everything in the world behaves like an Erlang process and communicate through message passing.

Each port has an owner, more on this later, but all processes who knows about the port can send messages to the port. In figure REF we see how a processes can communicate with the port and how the port is communicating to the world outside the Erlang node.



```
+-------------------+
|   __              |
|  /  \   owner    __         _______
|  \__/ < - -- -> /  \ <---->  /      /
|   P1            \__/        /______/
|              7    |Port1     file
|             /     |
|           /       |
|    __   /         |
|   /  \            |
```

```
  |   \__/             |
  |    P2              |
  +--------------------+
      Erlang Node

Process P1 has opened a port (Port1) to a file, and is the owner of
the port and can receive messages from the port. Process P2 also has a
handle to the port and can send messages to the port. The processes
and the port resides in an Erlang node. The file lives in the file and
operating system on the outside of the Erlang node.
```

If the port owner dies or is terminated the port is also killed. When a port terminates all external resources should also be cleaned up. This is true for all ports that comes with Erlang and if you implement your own port you should make sure it does this cleanup.

14.2.1 Different types of Ports

There are three different classes of ports: file descriptors, external programs and drivers. A file descriptor port makes it possible for a process to access an already opened file descriptor. A port to an external program invokes the external program as a separate OS process. A driver port requires a driver to be loaded in the Erlang node.

All ports are created by a call to `erlang:open_port(PortName, PortSettings)`.

A file descriptor port is opened with `{fd, In, Out}` as the `PortName`. This class of ports is used by some internal ERTS servers like the old shell. They are considered to not be very efficient and hence seldom used.

An external program port can be used to execute any program in the native OS of the Erlang node. To open an external program port you give either the argument `{spawn, Command}` or `{spawn_executable, FileName}` with the name of the external program. This is the easiest and one of the safest way to interact with code written in other programming languages. Since the external program is executed in its own OS process it will not bring down the Erlang node if it crashes. (It can of course use up all CPU or memory or do a number of other things to bring down the whole OS, but it is much safer than a linked in driver or a NIF.)

A driver port requires that a driver program has been loaded with ERTS. Such a port is started with either `{spawn, Command}` or `{spawn_driver, Command}`. Writing your own linked in driver can be an efficient way to interface for example some C library code that you would like to use. Note that a linked in driver executes in the same OS process as the Erlang node and a crash in the driver will bring down the whole node. Details about how to write an Erlang driver in general can be found in [?].

Erlang OTP comes with a number port drivers implementing the predefined port types. There are the common drivers available on all platforms: tcp_inet, udp_inet, sctp_inet, efile, zlib_drv, ram_file_drv, binary_filer, tty_sl. These drivers are used to implement e.g. file handling and sockets in Erlang. On Windows there is also a driver to access the registry: registry*drv*. And on most platforms there are example drivers to use when implementing your own driver like: multi and sig_drv.

```
+-------------------------------------------------------------------------------------------------
|                               Entities on an Erlang Node  ↩
                                               |
|--------------------------------------------------+----------------------------------------------
|                Processes                         |                    Ports  ↩
                           |
|--------------------------------------------------+---------++------++---------------------------
|                                                  | FD      ||  OS  ||         Drivers  ↩
                 |
|--------------------------------------------------+---------++------++----------+++-------+++-----
|                                                  | (Heart) || (ls) || tcp_inet ||| efile ↩
     ||| ...    |
+--------------------------------------------------+---------++------++----------+++-------+++-----
```

14.2.1.1 Ports to file descriptors

Creating

Commands

Examples

Implementation

14.2.1.2 Ports to Spawned OS Processes

Creating

Commands

Implementation

Windows

14.2.1.3 Ports to Linked in Drivers

Creating

Commands

Implementation

See chapter [?] for details of how to implement your own linked in driver.

14.3 Distributed Erlang

14.4 Sockets, UDP and TCP

Chapter 15

Distribution

Chapter 16

Interfacing C — BIFs NIFs and Linked in Drivers

Chapter 17

Native code

Part II

Running ERTS

Chapter 18

Operation and Maintenance

Part III

Apendicies

.1 Building Erlang OTP

.1 BEAM Instructions

Here we will go through most of the instructions in the BEAM generic instruction set in detail. In the next section we list all instructions with a brief explanation generated from the documentaion in the code (see `lib/compiler/src/genop.tab`).

.2 Functions and Labels

.2.1 label Lbl

Instruction number 1 in the generic instruction set is not really an instruction at all. It is just a module local label giving a name, or actually a number to the current position in the code.

Each label potentially marks the beginning of a basic block since it is a potential destination of a jump.

.2.2 func_info Module Function Arity

The code for each function starts with a func_info instruction. This instruction is used for generating a function clause error, and the execution of the code in the function actually starts at the label following the func_info instruction.

Imagine a function with a guard:

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

The Beam code for this function might look like:

```
{function, id, 1, 4}.
  {label,3}.
    {func_info,{atom,test1},{atom,id},1}.
  {label,4}.
    {test,is_integer,{f,3},[{x,0}]}.
    return.
```

Here the meta information `{function, id, 1, 4}` tells us that execution of the id/1 function will start at label 4. At label 4 we do an `is_integer` on x0 and if we fail we jump to label 3 (f3) which points to the func_info instruction, which will generate a *function clause* exception. Otherwise we just fall through and return the argument (x0).

.3 Test instructions

.3.1 Type tests

The type test instructions (`is_* Lbl Argument`) checks whether the argument is of the given type and if not jumps to the label Lbl. The beam disassembler wraps all these instructions in a `test` instruction. E.g.:

```
    {test,is_integer,{f,3},[{x,0}]}.
```

The current type test instructions are is_integer, is_float, is_number, is_atom, is_pid, is_reference, is_port, is_nil, is_binary, is_list, is_nonempty_list, is_function, is_function2, is_boolean, is_bitstr, and is_tuple.

And then there is also one type test instruction of Arity 3: `test_arity Lbl Arg Arity`. This instruction tests that the arity of the argument (assumed to be a tuple) is of `Arity`. This instruction is usually preceded by an `is_tuple` instruction.

.3.2 Comparisons

The comparison instructions (`is_* Lbl Arg1 Arg2`) compares the two arguments according to the instructions and jumps to Lbl if the comparison fails.

The comparison instructions are: : is_lt, is_ge, is_eq, is_ne, is_eq_exact, and is_ne_exact.

Remember that all Erlang terms are ordered so these instructions can compare any two terms. You can for example test if the atom `self` is less than the pid returned by `self()`. (It is.)

Note that for numbers the comparison is done on the Erlang type *number*, see Chapter 4. That is, for a mixed float and integer comparison the number of lower precision is converted to the other type before comparison. For example on my system 1 and 0.1 compares as equal, as well as 9999999999999999 and 1.0e16. Comparing floating point numbers is always risk and best avoided, the result may wary depending on the underlying hardware.

If you want to make sure that the integer 1 and the floating point number 1.0 are compared different you can use is_eq_exact and is_ne_exact. This corresponds to the Erlang operators `=:=` and `=/=`.

.4 Function Calls

In this chapter we will summarize what the different call instructions does. For a thorough description of how function calls work see Chapter 8.

.4.1 call Arity Label

Does a call to the function of arity `Arity` in the same module at label `Label`. First count down the reductions and if needed do a context switch.

For all local calls the label is the second label of the function where the code starts. It is assumed that the preceding instruction at that label is `func_info` in order to get the MFA if a context switch is needed.

.4.2 call_last Arity Label

Do a tail recursive call the function of arity `Arity` in the same module at label `Label`. First count down the reductions and if needed do a context switch.

.4.3 call_only Arity Label Deallocate

Deallocate `Deallocate` words of stack, then do a tail recursive call to the function of arity `Arity` in the same module at label `Label` First count down the reductions and if needed do a context switch.

.4.4 call_ext Arity Destination

Does an external call to the function of arity `Arity` given by Destination. Destination is usually of the form `{extfunc, Module, Function, Arity}`. First count down the reductions and if needed do a context switch.

.4.5 call_ext_only Arity Destination

Does a tail recursive external call to the function of arity `Arity` given by Destination. Destination is usually of the form `{extfunc, Module, Function, Arity}`. First count down the reductions and if needed do a context switch.

.4.6 call_ext_last Arity Destination Deallocate

Deallocate `Deallocate` words of stack, then do a tail recursive external call to the function of arity `Arity` given by Destination. Destination is usually of the form `{extfunc, Module, Function, Arity}`. First count down the reductions and if needed do a context switch.

.4.7 `bif0 Bif Reg, bif1 Lbl Bif Arg Reg, bif2 Lbl Bif Arg1 Arg2 Reg`

Call the bif `Bif` with the given arguments, and store the result in `Reg`. If the bif fails jump to `Lbl`. No zero arity bif can fail an thus those calls doesn't take a fail label.

.4.8 `gc_bif1-3 Lbl Live Bif Arg1-3 Reg`

Call the bif `Bif` with the given arguments, and store the result in Reg. If the bif fails jump to Lbl. Store the arguments in x(Live), x(Live+1) and x(live+2).

.4.9 `call_fun Arity`

The instruction `call_fun` assumes that the arguments are placed in the argument registers and that the fun (the pointer to the closure) is placed in the last argument register.

That is, for a zero arity call, the closure is in x0. For a arity 1 call x0 contains the argument and x1 contains the closure.

.4.10 apply/1

TODO

.4.11 apply_last/2

TODO

.5 Stack (and Heap) Management

The stack and the heap of an Erlang process on Beam share the same memory area see Chapter 3 and Chapter 12 for a full discussion. The stack grows toward lower addresses and the heap toward higher addresses. Beam will do a garbage collection if more space than what is available is needed on either the stack or the heap.

A leaf function A leaf function is a function which doesn't call any other function.

A non leaf function A non leaf function is a function which may call another function.

These instructions are also used by non leaf functions for setting up and tearing down the stack frame for the current instruction. That is, on entry to the function the *continuation pointer* (CP) is saved on the stack, and on exit it is read back from the stack.

A function skeleton for a leaf function looks like this:

```
{function, Name, Arity, StartLabel}.
  {label,L1}.
    {func_info,{atom,Module},{atom,Name},Arity}.
  {label,L2}.
    ...
    return.
```

A function skeleton for a non leaf function looks like this:

```
{function, Name, Arity, StartLabel}.
  {label,L1}.
    {func_info,{atom,Module},{atom,Name},Arity}.
  {label,L2}.
    {allocate,Need,Live}.

    ...
    call ...
    ...

    {deallocate,Need}.
    return.
```

.5.1 allocate StackNeed Live

Save the continuation pointer (CP) and allocate space for `StackNeed` extra words on the stack. If a GC is needed during allocation save `Live` number of X registers. E.g. if `Live` is 2 then registers X0 and X1 are saved.

When allocating on the stack, the stack pointer (E) is decreased.

Example .1 Allocate 1 0

```
       Before           After
         | xxx |            | xxx |
    E -> | xxx |            | xxx |
         |     |            | ??? | caller save slot
           ...         E -> | CP  |
           ...                ...
 HTOP -> |     |    HTOP -> |     |
         | xxx |            | xxx |
```

.5.2 allocate_heap StackNeed HeapNeed Live

Save the continuation pointer (CP) and allocate space for `StackNeed` extra words on the stack. Ensure that there also is space for HeapNeed words on the heap. If a GC is needed during allocation save `Live` number of X registers.

Note that the heap pointer (HTOP) is not changed until the actual heap allocation takes place.

.5.3 allocate_zero StackNeed Live

This instruction works the same way as allocate, but it also clears out the allocated stack slots with NIL.

Example .2 allocate_zero 1 0

```
       Before           After
         | xxx |            | xxx |
    E -> | xxx |            | xxx |
         |     |            | NIL | caller save slot
           ...         E -> | CP  |
           ...                ...
 HTOP -> |     |    HTOP -> |     |
         | xxx |            | xxx |
```

.5.4 allocate_heap_zero StackNeed HeapNeed Live

The allocate_heap_zero instruction works as the allocate_heap instruction, but it also clears out the allocated stack slots with NIL.

.5.5 test_heap HeapNeed Live

The test_heap instruction ensures there is space for HeapNeed words on the heap. If a GC is needed save Live number of X registers.

.5.6 init N

The init instruction clears N stack words. By writing NIL to them.

.5.7 deallocate N

The deallocate instruction is the opposite of the allocate instruction, it restores the continuation pointer and deallocates N+1 stack words.

.5.8 return

The return instructions jumps to the address in the continuation pointer (CP).

.5.9 trim N Remaining

Removes N words of stack usage, while keeping the continuation pointer on the top of the stack. (The argument Remaining is to the best of my knowledge unused.)

```
Trim 2
       Before            After
         | ??? |            | ??? |
         | xxx |       E -> | CP  |
         | xxx |            | ... |
    E -> | CP  |            | ... |
         |     |            | ... |
           ...                ...
 HTOP -> |     |    HTOP -> |     |
         | xxx |            | xxx |
```

.6 Moving, extracting, modifying.

.6.1 move Source Destination

Move the source Source (a litteral or a register) to the destination register Destination.

.6.2 get_list Source Head Tail

Get the head and tail (or car and cdr) parts of a list (a cons cell) from Source and put them into the registers Head and Tail.

.6.3 get_tuple_element Source Element Destination

Get element number Element from the tuple in Source and put it in the destination register Destination.

.6.4 set_tuple_element NewElement Tuple Position

Update the element at postition Position of the tuple Tuple with the new element NewElement.

.7 Building terms.

.7.1 put_list/3

TODO ==== put_tuple/2 TODO ==== put/1

.7.2 make_fun2/1

TODO

.8 Binary Syntax

.8.1 bs_put_integer/5

TODO ==== bs_put_binary/5 TODO ==== bs_put_float/5 TODO ==== bs_put_string/2 TODO ==== bs_init2/6 TODO ==== bs_add/5 TODO ==== bs_start_match2/5 TODO ==== bs_get_integer2/7 TODO ==== bs_get_float2/7 TODO ==== bs_get_binary2/7 TODO ==== bs_skip_bits2/5 TODO ==== bs_test_tail2/3 TODO ==== bs_save2/2 TODO ==== bs_restore2/2 TODO ==== bs_context_to_binary/1 TODO ==== bs_test_unit/3 TODO ==== bs_match_string/4 TODO ==== bs_init_writable/0 TODO ==== bs_append/8 TODO ==== bs_private_append/6 TODO ==== bs_init_bits/6 TODO ==== bs_get_utf8/5 TODO ==== bs_skip_utf8/4 TODO ==== bs_get_utf16/5 TODO ==== bs_skip_utf16/4 TODO ==== bs_get_utf32/5 TODO ==== bs_skip_utf32/4 TODO ==== bs_utf8_size/3 TODO ==== bs_put_utf8/3 TODO ==== bs_utf16_size/3 TODO ==== bs_put_utf16/3 TODO ==== bs_put_utf32/3 TODO

.9 Floating Point Arithmetic

.9.1 fclearerror/0

TODO ==== fcheckerror/1 TODO ==== fmove/2 TODO ==== fconv/2 TODO ==== fadd/4 TODO ==== fsub/4 TODO ==== fmul/4 TODO ==== fdiv/4 TODO ==== fnegate/3 TODO

.10 Pattern Matching

.10.1 select_val

TODO ==== select_arity_val TODO ==== jump TODO === Exception handling ==== catch/2 TODO ==== catch_end/1 TODO ==== badmatch/1 TODO ==== if_end/0 TODO ==== case_end/1 TODO

.11 Meta instructions

.11.1 on_load

TODO ==== line TODO

.12 Generic Instructions

Name	Arity	Op Code	Spec	Documentation
allocate	2	12	**allocate** *StackNeed*, *Live*	Allocate space for StackNeed words on the stack. If a GC is needed during allocation there are Live number of live X registers. Also save the continuation pointer (CP) on the stack.
allocate_heap	3	13	**allocate_heap** *StackNeed*, *HeapNeed*, *Live*	Allocate space for StackNeed words on the stack and ensure there is space for HeapNeed words on the heap. If a GC is needed save Live number of X registers. Also save the continuation pointer (CP) on the stack.
allocate_heap_zero	3	15	**allocate_heap_zero** *StackNeed*, *HeapNeed*, *Live*	Allocate space for StackNeed words on the stack and HeapNeed words on the heap. If a GC is needed during allocation there are Live number of live X registers. Clear the new stack words. (By writing NIL.) Also save the continuation pointer (CP) on the stack.

Name	Arity	Op Code	Spec	Documentation
allocate_zero	2	14	**allocate_zero** *StackNeed*, *Live*	Allocate space for StackNeed words on the stack. If a GC is needed during allocation there are Live number of live X registers. Clear the new stack words. (By writing NIL.) Also save the continuation pointer (CP) on the stack.
apply	1	112		
apply_last	2	113		
badmatch	1	72		
bif0	2	9	**bif0** *Bif*, *Reg*	Call the bif Bif and store the result in Reg.
bif1	4	10	**bif1** *Lbl*, *Bif*, *Arg*, *Reg*	Call the bif Bif with the argument Arg, and store the result in Reg. On failure jump to Lbl.
bif2	5	11	**bif2** *Lbl*, *Bif*, *Arg1*, *Arg2*, *Reg*	Call the bif Bif with the arguments Arg1 and Arg2, and store the result in Reg. On failure jump to Lbl.
bs_add	5	111		
bs_append	8	134		
bs_bits_to_bytes	3	(110)	**DEPRECATED**	
bs_bits_to_bytes2	2	(127)	**DEPRECATED**	
bs_context_to_binary	1	130		
bs_final	2	(88)	**DEPRECATED**	
bs_final2	2	(126)	**DEPRECATED**	
bs_get_binary	5	(82)	**DEPRECATED**	
bs_get_binary2	7	119		
bs_get_float	5	(81)	**DEPRECATED**	
bs_get_float2	7	118		
bs_get_integer	5	(80)	**DEPRECATED**	
bs_get_integer2	7	117		
bs_get_utf16	5	140		
bs_get_utf32	5	142		
bs_get_utf8	5	138		
bs_init	2	(87)	**DEPRECATED**	
bs_init2	6	109		
bs_init_bits	6	137		
bs_init_writable	0	133		
bs_match_string	4	132		
bs_need_buf	1	(93)	**DEPRECATED**	
bs_private_append	6	135		
bs_put_binary	5	90		
bs_put_float	5	91		
bs_put_integer	5	89		
bs_put_string	2	92		
bs_put_utf16	3	147		

Name	Arity	Op Code	Spec	Documentation
bs_put_utf32	3	148		
bs_put_utf8	3	145		
bs_restore	1	(86)	**DEPRECATED**	
bs_restore2	2	123		
bs_save	1	(85)	**DEPRECATED**	
bs_save2	2	122		
bs_skip_bits	4	(83)	**DEPRECATED**	
bs_skip_bits2	5	120		
bs_skip_utf16	4	141		
bs_skip_utf32	4	143		
bs_skip_utf8	4	139		
bs_start_match	2	(79)	**DEPRECATED**	
bs_start_match2	5	116		
bs_test_tail	2	(84)	**DEPRECATED**	
bs_test_tail2	3	121		
bs_test_unit	3	131		
bs_utf16_size	3	146		
bs_utf8_size	3	144		
call	2	4	**call** *Arity*, *Label*	Call the function at Label. Save the next instruction as the return address in the CP register.
call_ext	2	7	**call_ext** *Arity*, *Destination*	Call the function of arity Arity pointed to by Destination. Save the next instruction as the return address in the CP register.
call_ext_last	3	8	**call_ext_last** *Arity*, *Destination*, *Deallocate*	Deallocate and do a tail call to function of arity Arity pointed to by Destination. Do not update the CP register. Deallocate Deallocate words from the stack before the call.
call_ext_only	2	78	**call_ext_only** *Arity*, *Label*	Do a tail recursive call to the function at Label. Do not update the CP register.
call_fun	1	75	**call_fun** *Arity*	Call a fun of arity Arity. Assume arguments in registers x(0) to x(Arity-1) and that the fun is in x(Arity). Save the next instruction as the return address in the CP register.

Name	Arity	Op Code	Spec	Documentation
call_last	3	5	**call_last** *Arity*, *Label*, *Dellocate*	Deallocate and do a tail recursive call to the function at Label. Do not update the CP register. Before the call deallocate Deallocate words of stack.
call_only	2	6	**call_only** *Arity*, *Label*	Do a tail recursive call to the function at Label. Do not update the CP register.
case_end	1	74		
catch	2	62		
catch_end	1	63		
deallocate	1	18	**deallocate** *N*	Restore the continuation pointer (CP) from the stack and deallocate N+1 words from the stack (the + 1 is for the CP).
fadd	4	98		
fcheckerror	1	95		
fclearerror	0	94		
fconv	2	97		
fdiv	4	101		
fmove	2	96		
fmul	4	100		
fnegate	3	102		
fsub	4	99		
func_info	3	2	**func_info** *M*, *F*, *A*	Define a function M:F/A
gc_bif1	5	124	**gc_bif1** *Lbl*, *Live*, *Bif*, *Arg*, *Reg*	Call the bif Bif with the argument Arg, and store the result in Reg. On failure jump to Lbl. Do a garbage collection if necessary to allocate space on the heap for the result (saving Live number of X registers).
gc_bif2	6	125	**gc_bif2** *Lbl*, *Live*, *Bif*, *Arg1*, *Arg2*, *Reg*	Call the bif Bif with the arguments Arg1 and Arg2, and store the result in Reg. On failure jump to Lbl. Do a garbage collection if necessary to allocate space on the heap for the result (saving Live number of X registers).

Name	Arity	Op Code	Spec	Documentation
gc_bif3	7	152	**gc_bif3** *Lbl*, *Live*, *Bif*, *Arg1*, *Arg2*, *Arg3*, *Reg*	Call the bif Bif with the arguments Arg1, Arg2 and Arg3, and store the result in Reg. On failure jump to Lbl. Do a garbage collection if necessary to allocate space on the heap for the result (saving Live number of X registers).
get_list	3	65	**get_list** *Source*, *Head*, *Tail*	Get the head and tail (or car and cdr) parts of a list (a cons cell) from Source and put them into the registers Head and Tail.
get_map_elements	3	158		
get_tuple_element	3	66	**get_tuple_element** *Source*, *Element*, *Destination*	Get element number Element from the tuple in Source and put it in the destination register Destination.
has_map_fields	3	157		
if_end	0	73		
init	1	17	**init** *N*	Clear the Nth stack word. (By writing NIL.)
int_band	4	(33)	**DEPRECATED**	
int_bnot	3	(38)	**DEPRECATED**	
int_bor	4	(34)	**DEPRECATED**	
int_bsl	4	(36)	**DEPRECATED**	
int_bsr	4	(37)	**DEPRECATED**	
int_bxor	4	(35)	**DEPRECATED**	
int_code_end	0	3		
int_div	4	(31)	**DEPRECATED**	
int_rem	4	(32)	**DEPRECATED**	
is_atom	2	48	**is_atom** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not an atom.
is_binary	2	53	**is_binary** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a binary.
is_bitstr	2	129	**is_bitstr** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a bit string.
is_boolean	2	114	**is_boolean** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a Boolean.
is_constant	2	(54)	**DEPRECATED**	

Name	Arity	Op Code	Spec	Documentation
is_eq	3	41	**is_eq** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is not (numerically) equal to Arg2.
is_eq_exact	3	43	**is_eq_exact** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is not exactly equal to Arg2.
is_float	2	46	**is_float** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a float.
is_function	2	77	**is_function** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a function (i.e. fun or closure).
is_function2	3	115	**is_function2** *Lbl*, *Arg1*, *Arity*	Test the type of Arg1 and jump to Lbl if it is not a function of arity Arity.
is_ge	3	40	**is_ge** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is less than Arg2.
is_integer	2	45	**is_integer** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not an integer.
is_list	2	55	**is_list** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a cons or nil.
is_lt	3	39	**is_lt** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is not less than Arg2.
is_map	2	156		
is_ne	3	42	**is_ne** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is (numerically) equal to Arg2.
is_ne_exact	3	44	**is_ne_exact** *Lbl*, *Arg1*, *Arg2*	Compare two terms and jump to Lbl if Arg1 is exactly equal to Arg2.
is_nil	2	52	**is_nil** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not nil.
is_nonempty_list	2	56	**is_nonempty_list** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a cons.
is_number	2	47	**is_number** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a number.
is_pid	2	49	**is_pid** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a pid.

Name	Arity	Op Code	Spec	Documentation
is_port	2	51	**is_port** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a port.
is_reference	2	50	**is_reference** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a reference.
is_tuple	2	57	**is_tuple** *Lbl*, *Arg1*	Test the type of Arg1 and jump to Lbl if it is not a tuple.
jump	1	61	**jump** *Label*	Jump to Label.
label	1	1	**label** *Lbl*	Specify a module local label. Label gives this code address a name (Lbl) and marks the start of a basic block.
line	1	153		
loop_rec	2	23	**loop_rec** *Label*, *Source*	Loop over the message queue, if it is empty jump to Label.
loop_rec_end	1	24	**loop_rec_end** *Label*	Advance the save pointer to the next message and jump back to Label.
m_div	4	(30)	**DEPRECATED**	
m_minus	4	(28)	**DEPRECATED**	
m_plus	4	(27)	**DEPRECATED**	
m_times	4	(29)	**DEPRECATED**	
make_fun	3	(76)	**DEPRECATED**	
make_fun2	1	103		
move	2	64	**move** *Source*, *Destination*	Move the source Source (a literal or a register) to the destination register Destination.
on_load	0	149		
put	1	71		
put_list	3	69		
put_literal	2	(128)	**DEPRECATED**	
put_map_assoc	5	154		
put_map_exact	5	155		
put_string	3	(68)	**DEPRECATED**	
put_tuple	2	70		
raise	2	108		
recv_mark	1	150	**recv_mark** *Label*	Save the end of the message queue and the address of the label Label so that a recv_set instruction can start scanning the inbox from this position.

Name	Arity	Op Code	Spec	Documentation
recv_set	1	151	**recv_set** *Label*	Check that the saved mark points to Label and set the save pointer in the message queue to the last position of the message queue saved by the recv_mark instruction.
remove_message	0	21	**remove_message**	Unlink the current message from the message queue and store a pointer to the message in x(0). Remove any timeout.
return	0	19	**return**	Return to the address in the continuation pointer (CP).
select_tuple_arity	3	60	**select_tuple_arity** *Tuple*, *FailLabel*, *Destinations*	Check the arity of the tuple Tuple and jump to the corresponding destination label, if no arity matches, jump to FailLabel.
select_val	3	59	**select_val** *Arg*, *FailLabel*, *Destinations*	Jump to the destination label corresponding to Arg in the Destinations list, if no arity matches, jump to FailLabel.
send	0	20	**send**	Send argument in x(0) as a message to the destination process in x(0). The message in x(1) ends up as the result of the send in x(0).
set_tuple_element	3	67	**set_tuple_element** *NewElement*, *Tuple*, *Position*	Update the element at postition Position of the tuple Tuple with the new element NewElement.
test_arity	3	58	**test_arity** *Lbl*, *Arg1*, *Arity*	Test the arity of (the tuple in) Arg1 and jump to Lbl if it is not equal to Arity.
test_heap	2	16	**test_heap** *HeapNeed*, *Live*	Ensure there is space for HeapNeed words on the heap. If a GC is needed save Live number of X registers.
timeout	0	22	**timeout**	Reset the save point of the mailbox and clear the timeout flag.

Name	Arity	Op Code	Spec	Documentation
trim	2	136	**trim** *N*, *Remaining*	Reduce the stack usage by N words, keeping the CP on the top of the stack.
try	2	104		
try_case	1	106		
try_case_end	1	107		
try_end	1	105		
wait	1	25	**wait** *Label*	Suspend the processes and set the entry point to the beginning of the receive loop at Label.
wait_timeout	2	26	**wait_timeout** *Lable*, *Time*	Sets up a timeout of Time milllisecons and saves the address of the following instruction as the entry point if the timeout triggers.

.13 Specific Instructions

Argument types

Type	Explanation
t	A term, e.g. `[{foo,bar}]`
I	An integer e.g. `42`
x	A register, e.g. `5`
y	A stack slot, e.g. `1`
c	A constant (atom,nil,small int) // Pid?
a	An atom, e.g. *foo*
f	A label, i.e. a code address
s	Either a literal, a register or a stack slot
d	Either a register or a stack slot
r	A register R0
P	A positive (unsigned) integer literal
j	An optional code label
e	A reference to an export table entry
l	A floating-point register

List of all BEAM Instructions

Instruction	Arguments	Explanation
allocate	t t	
allocate_heap	t I t	
deallocate	I	
init	y	
init2	y y	
init3	y y y	
i_trim	I	
test_heap	I t	
allocate_zero	t t	
allocate_heap_zero	t I t	

Instruction	Arguments	Explanation
i_select_val	r f I	
i_select_val	x f I	
i_select_val	y f I	
i_select_val2	r f c f c f	
i_select_val2	x f c f c f	
i_select_val2	y f c f c f	
i_jump_on_val	rxy f I I	
i_jump_on_val_zero	rxy f I	
i_select_tuple_arity	r f I	
i_select_tuple_arity	x f I	
i_select_tuple_arity	y f I	
i_select_tuple_arity2	r f A f A f	
i_select_tuple_arity2	x f A f A f	
i_select_tuple_arity2	y f A f A f	
i_func_info	I a a I	
return		
get_list	rxy rxy rxy	
catch	y f	
catch_end	y	
try_end	y	
try_case_end	s	
set_tuple_element	s d P	
i_get_tuple_element	rxy P rxy	
is_number	f rxy	
jump	f	
case_end	rxy	
badmatch	rxy	
if_end		
raise	s s	
badarg	j	
system_limit	j	
move_jump	f ncxy	
move_x1	c	
move_x2	c	
move2	x y x y	
move2	y x y x	
move2	x x x x	
move	rxync rxy	
recv_mark	f	
i_recv_set	f	
remove_message		
timeout		
timeout_locked		
i_loop_rec	f r	
loop_rec_end	f	
wait	f	
wait_locked	f	
wait_unlocked	f	
i_wait_timeout	f I	
i_wait_timeout	f s	
i_wait_timeout_locked	f I	
i_wait_timeout_locked	f s	
i_wait_error		
i_wait_error_locked		
send		
i_is_eq_exact_immed	f rxy c	

Instruction	Arguments	Explanation
i_is_ne_exact_immed	f rxy c	
i_is_eq_exact_literal	f rxy c	
i_is_ne_exact_literal	f rxy c	
i_is_eq_exact	f	
i_is_ne_exact	f	
i_is_lt	f	
i_is_ge	f	
i_is_eq	f	
i_is_ne	f	
i_put_tuple	rxy I	
put_list	s s d	
i_fetch	s s	
move_return	xcn r	
move_deallocate_return	xycn r Q	
deallocate_return	Q	
test_heap_1_put_list	I y	
is_tuple_of_arity	f rxy A	
is_tuple	f rxy	
test_arity	f rxy A	
extract_next_element	xy	
extract_next_element2	xy	
extract_next_element3	xy	
is_integer_allocate	f rx I I	
is_integer	f rxy	
is_list	f rxy	
is_nonempty_list_allocate	f rx I t	
is_nonempty_list_test_heap	f r I t	
is_nonempty_list	f rxy	
is_atom	f rxy	
is_float	f rxy	
is_nil	f rxy	
is_bitstring	f rxy	
is_reference	f rxy	
is_pid	f rxy	
is_port	f rxy	
is_boolean	f rxy	
is_function2	f s s	
allocate_init	t I y	
i_apply		
i_apply_last	P	
i_apply_only		
apply	I	
apply_last	I P	
i_apply_fun		
i_apply_fun_last	P	
i_apply_fun_only		
i_hibernate		
call_bif0	e	
call_bif1	e	
call_bif2	e	
call_bif3	e	
i_get	s d	
self	rxy	
node	rxy	
i_fast_element	rxy j I d	
i_element	rxy j s d	

Instruction	Arguments	Explanation
bif1	f b s d	
bif1_body	b s d	
i_bif2	f b d	
i_bif2_body	b d	
i_move_call	c r f	
i_move_call_last	f P c r	
i_move_call_only	f c r	
move_call	xy r f	
move_call_last	xy r f Q	
move_call_only	x r f	
i_call	f	
i_call_last	f P	
i_call_only	f	
i_call_ext	e	
i_call_ext_last	e P	
i_call_ext_only	e	
i_move_call_ext	c r e	
i_move_call_ext_last	e P c r	
i_move_call_ext_only	e c r	
i_call_fun	I	
i_call_fun_last	I P	
i_make_fun	I t	
is_function	f rxy	
i_bs_start_match2	rxy f I I d	
i_bs_save2	rx I	
i_bs_restore2	rx I	
i_bs_match_string	rx f I I	
i_bs_get_integer_small_imm	rx I f I d	
i_bs_get_integer_imm	rx I I f I d	
i_bs_get_integer	f I I d	
i_bs_get_integer_8	rx f d	
i_bs_get_integer_16	rx f d	
i_bs_get_integer_32	rx f I d	
i_bs_get_binary_imm2	f rx I I I d	
i_bs_get_binary2	f rx I s I d	
i_bs_get_binary_all2	f rx I I d	
i_bs_get_binary_all_reuse	rx f I	
i_bs_get_float2	f rx I s I d	
i_bs_skip_bits2_imm2	f rx I	
i_bs_skip_bits2	f rx rxy I	
i_bs_skip_bits_all2	f rx I	
bs_test_zero_tail2	f rx	
bs_test_tail_imm2	f rx I	
bs_test_unit	f rx I	
bs_test_unit8	f rx	
bs_context_to_binary	rxy	
i_bs_get_utf8	rx f d	
i_bs_get_utf16	rx f I d	
i_bs_validate_unicode_retract	j	
i_bs_init_fail	rxy j I d	
i_bs_init_fail_heap	I j I d	
i_bs_init	I I d	
i_bs_init_heap	I I I d	
i_bs_init_heap_bin	I I d	
i_bs_init_heap_bin_heap	I I I d	
i_bs_init_bits_fail	rxy j I d	

Instruction	Arguments	Explanation
i_bs_init_bits_fail_heap	I j I d	
i_bs_init_bits	I I d	
i_bs_init_bits_heap	I I I d	
i_bs_add	j I d	
i_bs_init_writable		
i_bs_append	j I I I d	
i_bs_private_append	j I d	
i_new_bs_put_integer	j s I s	
i_new_bs_put_integer_imm	j I I s	
i_bs_utf8_size	s d	
i_bs_utf16_size	s d	
i_bs_put_utf8	j s	
i_bs_put_utf16	j I s	
i_bs_validate_unicode	j s	
i_new_bs_put_float	j s I s	
i_new_bs_put_float_imm	j I I s	
i_new_bs_put_binary	j s I s	
i_new_bs_put_binary_imm	j I s	
i_new_bs_put_binary_all	j s I	
bs_put_string	I I	
fmove	qdl ld	
fconv	d l	
i_fadd	l l l	
i_fsub	l l l	
i_fmul	l l l	
i_fdiv	l l l	
i_fnegate	l l l	
i_fcheckerror		
fclearerror		
i_increment	rxy I I d	
i_plus	j I d	
i_minus	j I d	
i_times	j I d	
i_m_div	j I d	
i_int_div	j I d	
i_rem	j I d	
i_bsl	j I d	
i_bsr	j I d	
i_band	j I d	
i_bor	j I d	
i_bxor	j I d	
i_int_bnot	j s I d	
i_gc_bif1	j I s I d	
i_gc_bif2	j I I d	
i_gc_bif3	j I s I d	
int_code_end		
label	L	
line	I	

.14 Full Code Listings

```
-module(beamfile).
-export([read/1]).
```

```
read(Filename) ->
    {ok, File} = file:read_file(Filename),
    <<"FOR1",
      Size:32/integer,
      "BEAM",
      Chunks/binary>> = File,
    {Size, parse_chunks(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)).

parse_chunks([{"Atom", _Size, <<_Numberofatoms:32/integer, Atoms/binary>>} | Rest], Acc) ->
    parse_chunks(Rest,[{atoms,parse_atoms(Atoms)}|Acc]);
parse_chunks([{"ExpT", _Size,
              <<_Numberofentries:32/integer, Exports/binary>>}
             | Rest], Acc) ->
    parse_chunks(Rest,[{exports,parse_table(Exports)}|Acc]);
parse_chunks([{"ImpT", _Size,
              <<_Numberofentries:32/integer, Imports/binary>>}
             | Rest], Acc) ->
    parse_chunks(Rest,[{imports,parse_table(Imports)}|Acc]);
parse_chunks([{"Code", Size, <<SubSize:32/integer, Chunk/binary>>} | Rest], Acc) ->
    <<Info:SubSize/binary, Code/binary>> = Chunk,
    OpcodeSize = Size - SubSize - 8, %% 8 is size of CunkSize & SubSize
    <<OpCodes:OpcodeSize/binary, _Align/binary>> = Code,
    parse_chunks(Rest,[{code,parse_code_info(Info), OpCodes}|Acc]);
parse_chunks([{"StrT", _Size, <<Strings/binary>>} | Rest], Acc) ->
    parse_chunks(Rest,[{strings,binary_to_list(Strings)}|Acc]);
parse_chunks([{"Attr", Size, Chunk} | Rest], Acc) ->
    <<Bin:Size/binary, _Pad/binary>> = Chunk,
    Attribs = binary_to_term(Bin),
    parse_chunks(Rest,[{attributes,Attribs}|Acc]);
parse_chunks([{"CInf", Size, Chunk} | Rest], Acc) ->
    <<Bin:Size/binary, _Pad/binary>> = Chunk,
    CInfo = binary_to_term(Bin),
    parse_chunks(Rest,[{compile_info,CInfo}|Acc]);
parse_chunks([{"LocT", _Size,
              <<_Numberofentries:32/integer, Locals/binary>>}
             | Rest], Acc) ->
    parse_chunks(Rest,[{locals,parse_table(Locals)}|Acc]);
parse_chunks([{"LitT", _ChunkSize,
              <<_CompressedTableSize:32, Compressed/binary>>}
             | Rest], Acc) ->
    <<_NumLiterals:32,Table/binary>> = zlib:uncompress(Compressed),
    Literals = parse_literals(Table),
    parse_chunks(Rest,[{literals,Literals}|Acc]);
parse_chunks([{"Abst", _ChunkSize, <<>>} | Rest], Acc) ->
    parse_chunks(Rest,Acc);
parse_chunks([{"Abst", _ChunkSize, <<AbstractCode/binary>>} | Rest], Acc) ->
    parse_chunks(Rest,[{abstract_code,binary_to_term(AbstractCode)}|Acc]);
parse_chunks([{"Line", _ChunkSize, <<LineTable/binary>>} | Rest], Acc) ->
    <<Ver:32,Bits:32,NumLineInstrs:32,NumLines:32,NumFnames:32,
      Lines:NumLines/binary,Fnames/binary>> = LineTable,
    parse_chunks(Rest,[{line,
                        [{version,Ver},
                         {bits,Bits},
```

```
                         {num_line_instrunctions,NumLineInstrs},
                         {lines,decode_lineinfo(binary_to_list(Lines),0)},
                         {function_names,Fnames}]}|Acc]);


parse_chunks([Chunk|Rest], Acc) -> %% Not yet implemented chunk
    parse_chunks(Rest, [Chunk|Acc]);
parse_chunks([],Acc) -> Acc.

parse_atoms(<<Atomlength, Atom:Atomlength/binary, Rest/binary>>) when Atomlength > 0->
    [list_to_atom(binary_to_list(Atom)) | parse_atoms(Rest)];
parse_atoms(_Alignment) -> [].

parse_table(<<Function:32/integer,
                Arity:32/integer,
                Label:32/integer,
                Rest/binary>>) ->
    [{Function, Arity, Label} | parse_table(Rest)];
parse_table(<<>>) -> [].


parse_code_info(<<Instructionset:32/integer,
                  OpcodeMax:32/integer,
                  NumberOfLabels:32/integer,
                  NumberOfFunctions:32/integer,
                  Rest/binary>>) ->
    [{instructionset, Instructionset},
     {opcodemax, OpcodeMax},
     {numberoflabels, NumberOfLabels},
     {numberofFunctions, NumberOfFunctions} |
     case Rest of
         <<>> -> [];
         _ -> [{newinfo, Rest}]
     end].

parse_literals(<<Size:32,Literal:Size/binary,Tail/binary>>) ->
    [binary_to_term(Literal) | parse_literals(Tail)];
parse_literals(<<>>) -> [].



-define(tag_i, 1).
-define(tag_a, 2).

decode_tag(?tag_i) -> i;
decode_tag(?tag_a) -> a.

decode_int(Tag,B,Bs) when (B band 16#08) =:= 0 ->
    %% N < 16 = 4 bits, NNNN:0:TTT
    N = B bsr 4,
    {{Tag,N},Bs};
decode_int(Tag,B,[]) when (B band 16#10) =:= 0 ->
    %% N < 2048 = 11 bits = 3:8 bits, NNN:01:TTT, NNNNNNNN
    Val0 = B band 2#11100000,
    N = (Val0 bsl 3),
    {{Tag,N},[]};
decode_int(Tag,B,Bs) when (B band 16#10) =:= 0 ->
    %% N < 2048 = 11 bits = 3:8 bits, NNN:01:TTT, NNNNNNNN
    [B1|Bs1] = Bs,
    Val0 = B band 2#11100000,
    N = (Val0 bsl 3) bor B1,
    {{Tag,N},Bs1};
```

```
decode_int(Tag,B,Bs) ->
    {Len,Bs1} = decode_int_length(B,Bs),
    {IntBs,RemBs} = take_bytes(Len,Bs1),
    N = build_arg(IntBs),
    {{Tag,N},RemBs}.

decode_lineinfo([B|Bs], F) ->
    Tag = decode_tag(B band 2#111),
    {{Tag,Num},RemBs} = decode_int(Tag,B,Bs),
    case Tag of
        i ->
            [{F, Num} | decode_lineinfo(RemBs, F)];
        a ->
            [B2|Bs2] = RemBs,
            Tag2 = decode_tag(B2 band 2#111),
            {{Tag2,Num2},RemBs2} = decode_int(Tag2,B2,Bs2),
            [{Num, Num2} | decode_lineinfo(RemBs2, Num2)]
    end;
decode_lineinfo([],_) -> [].

decode_int_length(B, Bs) ->
    {B bsr 5 + 2, Bs}.


take_bytes(N, Bs) ->
    take_bytes(N, Bs, []).

take_bytes(N, [B|Bs], Acc) when N > 0 ->
    take_bytes(N-1, Bs, [B|Acc]);
take_bytes(0, Bs, Acc) ->
    {lists:reverse(Acc), Bs}.


build_arg(Bs) ->
    build_arg(Bs, 0).

build_arg([B|Bs], N) ->
    build_arg(Bs, (N bsl 8) bor B);
build_arg([], N) ->
    N.
```

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

-include("world.hrl").

hello() -> ?GREETING.
```

```
-module(json_parser).
-export([parse_transform/2]).

parse_transform(AST, _Options) ->
    json(AST, []).

-define(FUNCTION(Clauses), {function, Label, Name, Arity, Clauses}).

%% We are only interested in code inside functions.
json([?FUNCTION(Clauses) | Elements], Res) ->
    json(Elements, [?FUNCTION(json_clauses(Clauses)) | Res]);
json([Other|Elements], Res) -> json(Elements, [Other | Res]);
json([], Res) -> lists:reverse(Res).
```

```
%% We are interested in the code in the body of a function.
json_clauses([{clause, CLine, A1, A2, Code} | Clauses]) ->
    [{clause, CLine, A1, A2, json_code(Code)} | json_clauses(Clauses)];
json_clauses([]) -> [].


-define(JSON(Json), {bin, _, [{bin_element
                                          , _
                                          , {tuple, _, [Json]}
                                          , _
                                          , _}]}).

%% We look for: <<"json">> = Json-Term
json_code([])                     -> [];
json_code([?JSON(Json)|MoreCode]) -> [parse_json(Json) | json_code(MoreCode)];
json_code(Code)                   -> Code.

%% Json Object -> [{}] | [{Lable, Term}]
parse_json({tuple,Line,[]})            -> {cons, Line, {tuple, Line, []}};
parse_json({tuple,Line,Fields})        -> parse_json_fields(Fields,Line);
%% Json Array -> List
parse_json({cons, Line, Head, Tail})   -> {cons, Line, parse_json(Head),
                                                       parse_json(Tail)};
parse_json({nil, Line})                -> {nil, Line};
%% Json String -> <<String>>
parse_json({string, Line, String})     -> str_to_bin(String, Line);
%% Json Integer -> Intger
parse_json({integer, Line, Integer})   -> {integer, Line, Integer};
%% Json Float -> Float
parse_json({float, Line, Float})       -> {float, Line, Float};
%% Json Constant -> true | false | null
parse_json({atom, Line, true})         -> {atom, Line, true};
parse_json({atom, Line, false})        -> {atom, Line, false};
parse_json({atom, Line, null})         -> {atom, Line, null};

%% Variables, should contain Erlang encoded Json
parse_json({var, Line, Var})         -> {var, Line, Var};
%% Json Negative Integer or Float
parse_json({op, Line, '-', {Type, _, N}}) when Type =:= integer
                                                ; Type =:= float ->
                                          {Type, Line, -N}.
%% parse_json(Code)                  -> io:format("Code: ~p~n",[Code]), Code.

-define(FIELD(Lable, Code), {remote, L, {string, _, Label}, Code}).

parse_json_fields([], L) -> {nil, L};
%% Label : Json-Term  --> [{<<Label>>, Term} | Rest]
parse_json_fields([?FIELD(Lable, Code) | Rest], _) ->
    cons(tuple(str_to_bin(Label, L), parse_json(Code), L)
         , parse_json_fields(Rest, L)
         , L).


tuple(E1, E2, Line)    -> {tuple, Line, [E1, E2]}.
cons(Head, Tail, Line) -> {cons, Line, Head, Tail}.

str_to_bin(String, Line) ->
    {bin
     , Line
     , [{bin_element
         , Line
```

```
         , {string, Line, String}
         , default
         , default
        }
       ]
    }.
```

```
-module(json_test).
-compile({parse_transform, json_parser}).
-export([test/1]).

test(V) ->
    <<{{
      "name"  : "Jack (\"Bee\") Nimble",
      "format": {
                  "type"      : "rect",
                  "widths"     : [1920,1600],
                  "height"    : (-1080),
                  "interlace" : false,
                  "frame rate": V
                }
     }}>>.
```

```
-module(stack_machine_compiler).
-export([compile/2]).

compile(Expression, FileName) ->
    [ParseTree] = element(2,
                          erl_parse:parse_exprs(
                            element(2,
                                    erl_scan:string(Expression)))),
    file:write_file(FileName, generate_code(ParseTree) ++ [stop()]).

generate_code({op, _Line, '+', Arg1, Arg2}) ->
    generate_code(Arg1) ++ generate_code(Arg2) ++ [add()];
generate_code({op, _Line, '*', Arg1, Arg2}) ->
    generate_code(Arg1) ++ generate_code(Arg2) ++ [multiply()];
generate_code({integer, _Line, I}) -> [push(), integer(I)].

stop()      -> 0.
add()       -> 1.
multiply() -> 2.
push()      -> 3.
integer(I) ->
    L = binary_to_list(binary:encode_unsigned(I)),
    [length(L) | L].
```

```
#include <stdio.h>
#include <stdlib.h>

char *read_file(char *name) {
  FILE *file;
  char *code;
  long  size;

  file = fopen(name, "r");

  if(file == NULL) exit(1);

  fseek(file, 0L, SEEK_END);
  size = ftell(file);
```

```
  code = (char*)calloc(size, sizeof(char));
  if(code == NULL) exit(1);

  fseek(file, 0L, SEEK_SET);

  fread(code, sizeof(char), size, file);
  fclose(file);
  return code;
}

#define STOP 0
#define ADD  1
#define MUL  2
#define PUSH 3

#define pop()   (stack[--sp])
#define push(X) (stack[sp++] = X)

int run(char *code) {
  int stack[1000];
  int sp = 0, size = 0, val = 0;
  char *ip = code;

  while (*ip != STOP) {
    switch (*ip++) {
    case ADD: push(pop() + pop()); break;
    case MUL: push(pop() * pop()); break;
    case PUSH:
      size = *ip++;
      val = 0;
      while (size--) { val = val * 256 + *ip++; }
      push(val);
      break;
    }
  }
  return pop();
}


int main(int argc, char *argv[])
{
  char *code;
  int res;

  if (argc > 1) {
    code = read_file(argv[1]);
    res = run(code);
    printf("The value is: %i\n", res);
    return 0;
  } else {
    printf("Give a the file name of a byte code program as argument\n");
    return -1;
  }
}
```

```
#include <stdio.h>
#include <stdlib.h>

#define STOP 0
#define ADD  1
#define MUL  2
#define PUSH 3
```

```
#define pop() (stack[--sp])
#define push(X) (stack[sp++] = (X))

typedef void (*instructionp_t)(void);

int stack[1000];
int sp;
instructionp_t *ip;
int running;

void add()  { int x,y; x = pop(); y = pop(); push(x + y); }
void mul()  { int x,y; x = pop(); y = pop(); push(x * y); }
void pushi(){ int x;   x = (int)*ip++;       push(x); }
void stop() { running = 0; }

instructionp_t *read_file(char *name) {
  FILE *file;
  instructionp_t *code;
  instructionp_t *cp;
  long  size;
  char ch;
  unsigned int val;

  file = fopen(name, "r");

  if(file == NULL) exit(1);

  fseek(file, 0L, SEEK_END);
  size = ftell(file);
  code = calloc(size, sizeof(instructionp_t));
  if(code == NULL) exit(1);
  cp = code;

  fseek(file, 0L, SEEK_SET);
  while ( ( ch = fgetc(file) ) != EOF )
    {
      switch (ch) {
      case ADD: *cp++ = &add; break;
      case MUL: *cp++ = &mul; break;
      case PUSH:
        *cp++ = &pushi;
        ch = fgetc(file);
        val = 0;
        while (ch--) { val = val * 256 + fgetc(file); }
        *cp++ = (instructionp_t) val;
        break;
      }
    }
  *cp = &stop;

  fclose(file);
  return code;
}


int run() {
  sp = 0;
  running = 1;

  while (running) (*ip++)();
```

```
  return pop();
}


int main(int argc, char *argv[])
{
  if (argc > 1) {
    ip = read_file(argv[1]);
    printf("The value is: %i\n", run());
    return 0;
  } else {
    printf("Give a the file name of a byte code program as argument\n");
    return -1;
  }
}
```

```
-module(share).

-export([share/2, size/0]).

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

size() ->
    T = share:share(5,[a,b,c]),
    {{size, erts_debug:size(T)},
     {flat_size, erts_debug:flat_size(T)}}.
```

```
-module(send).
-export([test/0]).

test() ->
    P2 = spawn(fun() -> p2() end),
    P1 = spawn(fun() -> p1(P2) end),
    {P1, P2}.

p2() ->
    receive
        M -> io:format("P2 got ~p", [M])
    end.

p1(P2) ->
    L = "hello",
    M = {L, L},
    P2 ! M,
    io:format("P1 sent ~p", [M]).
```

Load Balancer.

```
-module(lb).
-export([start/0]).

start() ->
    Workers = [spawn(fun worker/0) || _ <- lists:seq(1,10)],
    LoadBalancer = spawn(fun() -> loop(Workers, 0) end),
    {ok, Files} = file:list_dir("."),
    Loaders = [spawn(fun() -> loader(LoadBalancer, F) end) || F <- Files],
    {Loaders, LoadBalancer, Workers}.

loader(LB, File) ->
    case  file:read_file(File) of
```

```
        {ok, Bin} ->  LB ! Bin;
        _Dir -> ok
    end,
    ok.

worker() ->
    receive
        Bin ->
            io:format("Byte Size: ~w~n", [byte_size(Bin)]),
            garbage_collect(),
            worker()
    end.


loop(Workers, N) ->
  receive
    WorkItem ->
       Worker = lists:nth(N+1, Workers),
       Worker ! WorkItem,
       loop(Workers, (N+1) rem length(Workers))
  end.
```

Chapter 1

Index

B
BEAM, 4

E
Erlang RunTime System, 2
Erlang Runtime System, 2
Erlang Virtual Machine, 4
ERTS, 2, 3
EVM
    see=Erlang Virtual Machine|hyperpage, 4

N
node, 3

O
OTP, 3

S
see=Erlang Virtual Machine|hyperpage, 4
SMP, 5