

Overview of Ernovos

An operating system in Erlang

Erik Reitsma

May 3, 2023

Objectives

- As little C code as possible
- As much in Erlang as possible
- No use of C code of others
- Support 32-bit x86 (i586)
- Support ARM (Zynq)

Personal objectives

- Learn about operating systems
- Learn about Erlang/BEAM
- Learn about C programming
- Have fun!

Some Erlang: modules and recursion

```
-module(myfun).
```

```
-export([fac/1]).
```

```
fac(0) -> 1;
```

```
fac(N) -> N * fac(N - 1).
```

Some Erlang: modules and recursion

```
-module(myfun).
```

```
-export([fac/1]).
```

```
fac(N) when N < 1 -> 1;
```

```
fac(N) -> N * fac(N - 1).
```

Some Erlang: spawning processes and sending messages

```
F = fun() ->
    receive
        {print, Message} ->
            io:format("Got message: ~p~n", [Message]);
        stop ->
            ok
    end
end,
P = spawn(F),
P ! {print, "Hello World!"}.
```

Some Erlang: pattern matching with bitstrings

```
Eshell V12.3.2.1 (abort with ^G)
1> A = <<100000:32/big>>.
<<0,1,134,160>>
2> <<B1:16, B2:4, B3:4, B4:8>> = A.
<<0,1,134,160>>
3> [B1, B2, B3, B4].
[1,8,6,160]
```

Decoding an Ethernet packet

```
decode_frame(<<Destination:6/binary,  
             Source:6/binary,  
             EtherType:16/big,  
             Payload/binary>>) ->  
#ethernet_frame{destination = Destination,  
                 source = Source,  
                 type = EtherType,  
                 payload = Payload}.
```

Decoding an IP packet

```
decode(<<Version:4, Header_length:4,  
      Differentiated_Services:8, Total_length:16,  
      Identification:16, Flags:3,  
      Fragment_offset:13, Time_to_live:8,  
      Protocol:8, Header_Checksum:16,  
      Source_Address:4/binary, Destination_Address:4/binary,  
      OptionsAndData/binary>>) ->  
OptionsSize = 4 * (Header_length - 5),  
<<Options:OptionsSize/binary, DataAndMore/binary>> = OptionsAndData,  
%% sometimes we may get too much data to be safe,  
%% try to cut it off, if we can  
DataLength = Total_length - (Header_length * 4),  
...
```

Some Erlang: lists and tuples

```
1> A = "Hello World".  
"Hello World"  
2> is_list(A).  
true  
3> hd(A).  
72  
4> B = [5000 | A].  
[5000,72,101,108,108,111,32,87,111,114,108,100]  
5> T = {A, B}.  
{ "Hello World",  
  [5000,72,101,108,108,111,32,87,111,114,108,100]}  
6> {X, Y} = T.  
{ "Hello World",  
  [5000,72,101,108,108,111,32,87,111,114,108,100]}  
7> tl(Y).  
"Hello World"
```

Separate layers

- Hardware dependent (i586, zynq)
- Low level OS (LLOS)
- Erlang virtual machine (ErI VM)
- Erlang code (ErI OS)

Functions specific for i586

- Bootloader (primary and secondary)
- Utilities for disk image creation
- Inventory of memory
- UART (for debugging native code)
- Installing of fault handlers
- Installing of interrupt handlers
- Installing of timer (PIT)

Functions specific for Zynq

- Registration of configured free memory
- Installing of interrupt handlers
- Setup of TTC timer
- Cache configuration

Low level OS (LLOS)

Generic functions for use by the Erlang Virtual machine

- Management of memory pages
- `malloc()`
- `memset()`, `memmove()` etc.
- `printf()` to UART and VGA (if enabled)

- Large allocations get pages
- Small allocations taken from bins
- Bins have linked lists of blocks
- Block may span several pages
- Block contains items of equal size
- Block has header with info on item size
- Binfo specifies item size, number of items per block, number of pages, header size
- Header has bitmap of used items in block
- Four bytes before allocated memory contain information about type and size

Erlang Virtual Machine

Basic features of the virtual machine

- Adapted BEAM modules
- Data types
- Processes (many, no shared data)
- Messages
- Garbage collection per process
- Erlang functions implemented in C
- Limited CPU stack, but use `term_stack` in pages
- 32-bit aligned

Additions for hardware handling

- Read and write memory addresses
- Allocation of memory buffers
- Interrupt handling
- Timer handling (for receive after)

Instructions from `.beam` files are implemented directly

- Each instruction in a separate file
- Instructions up to and including OTP 21
- Instruction loop is in a single function
- Current state is in local variables
- Use `goto` to jump to next instruction
- Instruction addresses are put in `.beam` file during loading

Opcode: move

```
Operation(move)
{
    erl_term value;

    kdebug_instruction("move", 2);

    ArgGet(Argument(1), value);
    argSet(current_process, Argument(2), value);

    NextInstruction(2);
    CheckAndContinue;
}
```

Code loading

- Original .beam-file converted for easier processing
- Initial code loaded from tar in VM binary
- Copied to allocated memory
- Opcodes replaced by location of instruction implementation
- Atoms added to atom table
- Labels replaced by location of referenced code
- Module versions prepared

Term fits in 32 bits

- Pointer to heap type (prefix 0b00)
- Small int (prefix 0b01)
- Cons cell (prefix 0b10)
- Atom (index) (prefix 0b00011)
- Proces (prefix 0b00111)

Heap types

Term has prefix 0b00 and points to process heap

- Tuple
- Shared binary
- Large int
- Function
- Local binary
- Float
- Binary context (used while matching binaries)
- Reference
- Map
- Membuf (added for drivers)
- External function

Large integers

- Stored in array of 32 bits unsigned int
- Stored on process heap
- Header contains size and sign
- Arithmetic: add, subtract, multiply, divide, remainder, negate, compare
- Bitwise logical: and, or, not, xor
- Combinations large/small integers

- Local binaries on process heap
- Shared binaries in `malloced` blob
- Header specifies size and bit offset
- Local binaries are copied
- Shared binaries are reference counted by process
- Reference counts are updated on process exit
- Process remembers seen shared binaries

Lambda functions

- Function is generated by compiler in defining module
- Module name and function index in struct on heap
- Relevant bound variables in struct on heap
- Just another value on heap
- Bound variables will be copied when sent as message

Memory buffers for drivers

- Allocate memory
- Optional number of alignment bits
- Get address of membuf
- Write binary to membuf
- Read binary from membuf
- Owned by creating process
- Only owner process can use the membuf

Process state in struct `erl_process` (53 members)

- Current opcode
- Registers
- Message queue
- Heap
- Call stack
- Catch stack
- Shared binaries
- Membufs
- Process dictionary

- Instruction limit per process
- Process is also descheduled on `wait` instruction
- Local variables stored in `erl_process`
- Next runnable process selected
- Process is runnable if it was scheduled out or has a waiting message

Error handling

- Error cause set in local variable
- Error cause checked after each instruction
- Catch stack in process state unwound
- Linked and monitoring processes informed on process termination

Built-in functions (BIFs)

- Implemented inline
- Opcodes for different arities
- Implemented in VM in `switch` statements
- Registers and local state directly accessible
- Should be fast
- Continue with next instruction

Native implemented functions (NIFs)

- Stubs in Erlang modules
- Overwrite location of function with function pointer native implementation
- Access to memory and special instructions, interrupt tables
- `<module_name>_module_create.c` registers functions
- `<module_name>_module_impl.c` implements function
- Continue with Return

Example of native implemented function

```
BifImplementation(x86_outWord_2) {  
    if (is_small_int(current_process->regX[0])) {  
        if (is_small_int(current_process->regX[1])) {  
            outWord((uint16_t)term_to_small_int(current_process->regX[0]),  
                    (uint16_t)term_to_small_int(current_process->regX[1]));  
            setX(current_process, 0, ATOM_ok);  
        } else {  
            fault = ATOM_badarg;  
        }  
    } else {  
        fault = ATOM_badarg;  
    }  
    Return;  
}
```

Garbage collection per process

- Copy all live data to new allocated memory
- Live data is in process registers, stack, process dictionary, messages
- Pointer locations will change
- Register local term variables with collector to be updated
- After copy, old heap range is freed

Interrupt handling

- Process can register for interrupt
- Process can register service
- Service is script with defined instructions
- Load, move, store, and, or, xor, not, add, sub
- Input, output (Intel)
- Send message to process

Interrupt handling example: keyboard

```
interrupt_server:add_driver(1, self(),  
    [{move_immediate, 16#60, 0},  
     {in, byte, 0, 1},  
     {send, 1}])
```

- Register this process for IRQ 1
- Write 0x60 to register 0
- Read from input port register 0 into register 1
- Send value of register 1 to process

- Assembly code
- Primary loader from sector 0 loads secondary loader
- Secondary enables A20
- Reads memory map to provide to C memory page handler
- Reads boot device from BIOS
- Loads ELF header from hard coded location using BIOS
- Loads ELF binary using BIOS
- Prepares and enters protected mode

- Initialize UART
- Process boot information from bootloader
- Initialize terminal
- Install interrupt and fault handlers
- Initialize timer

Virtual machine start

- Initialize atom table
- Initialize shared heap
- Initialize opcode table
- Load modules from tar
- Initialize BIF modules
- Create runtime
- Create first process
- Initialize and enable interrupts
- Jump into runtime loop

Start of Erlang code (x86)

- Start Erlang term storage (ets)
- Start timer service
- Start init process
- Start logger
- Start interrupt server
- Start device manager
- Start file server
- Start mount manager
- Start rtc driver
- Start VGA text driver
- Start network manager
- Start PCI handler
- Mount first ATA device as root of file system
- Start keyboard driver

Start of Erlang code (Zynq)

- Start Erlang term storage (ets)
- Start init process
- Start Zynq MMU
- Start interrupt server
- Start Zynq GPIO handler
- Enable some GPIO pins for output (LEDs)
- Start network manager
- Start Zynq Ethernet driver
- Start Zynq UART server
- Start TTY on UART
- Blink some LEDs

Features implemented in Erlang

- Hardware drivers
- File systems
- Networking
- Shell on terminal
- Crypto

Hardware drivers in Erlang

- VGA
- PCI enumeration
- PCI generic ATA
- Intel EtherExpress PRO/100 Ethernet (QEmu)

- Generic block API
- Generic ATA implements block API
- FAT uses block API
- FAT read-only
- FAT implements low-level file API in OTP

- Ethernet driver
- `ethernet_handler`
- ARP
- IPv4, some IPv6
- TCP, UDP, ICMP
- DHCP client
- Implementation of OTP `gen_tcp` and `gen_udp`
- Minimal HTTP server
- SSH server from OTP

- Implementation of functions from OTP module `crypto` in Erlang
- AES128, AES192, AES256
- Modes CBC, CFB (block), CTR (stream)
- Elliptic curves
- Prime and polynomial based fields
- SHA224, SHA256, SHA384, SHA512
- ECDSA, ECDH
- Sufficient for SSH server

- Page tables
- 64 bit (instructions, data structures)
- New BEAM instructions
- RISC-V
- USB
- Tests