How to Build an OS in Erlang:
A Whistle-stop Tour of HydrOS

Sam Williams

University of Kent

June 9, 2017
Outline

1. Why Build an Erlang OS?
2. Existing Systems and Architectures
4. Building An Erlang OS: Challenges
5. General Lessons Learnt
6. Delving Into HydrOS
7. Demo
8. Acknowledgements
Why Build an Erlang OS?

- Fault tolerance
- Scalability
- Native high-level execution environment with universal data exchange format (even including complex terms like functions).
- Machine independent programming environment – run the same OS and program code on x86, ARM, etc.
Erlang Unikernel Hosting Systems

ErlangOnXen

- Uses the Ling VM.
- Targets paravirtualised Xen deployments (among others).
- The most complete Erlang unikernel solution.
- 4mb deployable image sizes, with sub 250ms boot times.
Erlang on RumpRun

- BEAM on small ‘rump’ kernel.
- Generates 6mb images.

Erlang on OSv

- Erjang on OSv, a small linux-compatible OS.
- Generates ‘fat’ unikernels.
- Greater Linux compatibility, at the cost of image deployment size.

ErlOS

- BEAM (on MiniOS) on Xen.
- Proof of concept. No longer supported.
Embedded Erlang Platforms

GRiSP
- Erlang on RTEMS (a small real-time OS).
- Built as a platform for creating wireless IoT applications.
- Targets ARM.

NERVES Project
- Elixir/Erlang on a thin Linux layer.
- Working to expose kernel functionality within the BEAM.

Erlang Embedded Initiative
- erlang-mini packages for embedded devices.
- Actor Library for Embedded (ALE).
A general purpose operating system for server and desktop systems.

Focuses on providing fault-tolerance and error recovery for typically catastrophic OS and hardware events.

Written almost entirely in Erlang – from inter-node message passing and drivers, to GUI applications.
Open Areas in Erlang OS Design

- Uninterpreted Erlang code on bare-metal via HiPE.
  - One unikernel per process?
  - Unikernels that build and launch other unikernels when processes are spawned?

- A tiny co-operative Erlang VM for many-core embedded devices (for example, the Paralella).
  - Support for platforms with hardware message passing?
Building An Erlang OS: Challenges

- How will your Erlang code run?
  - Will it be ‘native’, interpreted, or interpreted within another VM?

- Will it be BEAM compatible?
  - At what level? Instructions, AST, or BEAM code transpiler?
Erlang OS System Architectures

- Erlang Code
  - Hardware
  - Erlang On Metal

- Erlang Code
  - VM
  - Hardware
  - VM OS

- Erlang Code
  - VM
  - VM
  - VM
  - VM
  - Hardware
  - Multikernel OS
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Erlang Code</th>
<th>VM</th>
<th>Hypervisor</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unikernel Erlang</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Erlang Code</th>
<th>VM</th>
<th>VM</th>
<th>VM</th>
<th>VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Unikernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erlang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypervisor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architecture</th>
<th>App</th>
<th>C Code</th>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiunikernel Erlang</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erlang Code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How much of the work of the OS will be performed in Erlang? How much will be performed by native code?

- Will drivers be written in Erlang?
- What about performance critical GUI app code?
- Will you expose an Erlang interface to malloc and free, allowing raw buffers?

HydrOS uses a system of layers:

- Local kernel layer
- Local OS layer
- Global OS layer
Language Responsibility Division in HydrOS

Erlang Code

Global OS Layer

Inter-Processor Communication

Local OS Layer

Interrupts

BIFs

Local Kernel

C Code

How to Build an OS in Erlang: A Whistle-stop Tour of HydrOS

Sam Williams (University of Kent)
Foreign Function Interfaces

How will your operating system incorporate native (C, assembly) code, if at all?

- How will you load and execute libraries?
- If you are using them, how will your native-code drivers interact with the VM scheduling system? Could they be purely event (interrupt) driven?
Reaching a Secure Steady-State

How will your OS boot?

- Will you use an existing bootloader like GRUB/LILO/SYSLINUX?
- Will you implement EFI boot?

Once the OS has loaded, how will you ensure its security?

- Secure the perimeter, not the interior?
- HydrOS style capabilities?
General Lessons Learnt

- There is a fault-tolerance–performance spectrum on which your OS must be placed.
  - Native-code is fast, but failures are harder to recover from.
  - This trade-off is particularly important for drivers and interrupt handlers.
- You may not need SMP support – design your systems appropriately.
  - ‘This simplifies the implementation greatly and speeds things up. The philosophy is that you need more VMs to achieve true multi-core parallelism. Hypervisor is the only ‘hardware’ scheduler, Erlang processes are green threads.’
  - Maxim Kharchenko, ErlangOnXen.
Erlang OS design, by example.

- Architecture
- Implementation
- Demonstration
HydrOS is built with a multikernel architecture.

- Split the machine into multiple independent cores.
- Each core gets a VM.
- ‘Single System Image’ layer unifies environment at Erlang level.

![Multikernel Architecture Diagram](diagram.png)

Sam Williams (University of Kent) How to Build an OS in Erlang:A Whistle-stop Tour of HydrOS June 9, 2017 18 / 34
Fault-Tolerance Through Isolation

- Failures (even at the hardware level) in one core will not affect other cores.

- OS subsystems and drivers are also isolated from one another.
  - Restartable on demand.

- Potential for de-centralisation in the future.
HydrOS uses a library of BIFs to interact directly with the hardware.

- Provides interfaces to CPU functionality.
  - Enabling and disabling interrupts, for example.
- (Roughly) 24 functions, most with very short definitions.
HydrOS currently uses Erlang-only drivers.

- Sets of communicating processes that provide message-passing interfaces for hardware interaction.
- Can (should?) be spread across multiple nodes in a machine.
- IOAPICs used to route interrupts to the correct core.
- De-asserts interrupts after message is generated.
  - May be a problem for level-triggered interrupts.
Example HydrOS Driver

- **Interrupt Dispatcher**
- **Keyboard Handler**
- **Keyboard Listener**
- **Node Kernel**

Int. 0x21 events are handled by the Interrupt Dispatcher, which routes them to the Keyboard Handler. The Keyboard Handler then dispatches the events to the Keyboard Listeners.
HydrOS provides a framework for building and organising graphical terminal applications.

- Composable window interfaces.
- Necessarily multi-process apps.
- Apps can be distributed across different Erlang VMs, but present as if they are part of a single operating system.
- Currently uses raw memory buffers.
A Generic HydrOS WM App

Keyboard Driver → Key Events → App Input Proc. → Render Commands → wm_window

If foregrounded, Draw to Output

VGA Text Buffer
How to Build an OS in Erlang: A Whistle-stop Tour of HydrOS

Sam Williams (University of Kent)
The HydrOS approach to native code hosting.

- Dedicate a single core in the system to achieving your task.
- Run a unikernel program on this core directly, without intervention by any other part of the system.
- Provide a library for communication with Erlang nodes.

HydrOS MUKs can be created by simply placing a C file in a directory in the source tree.

The MUK generation system also accommodates more complex build environments.
Create a tree of decreasingly capable processes.

Processes have the same or fewer capabilities than their parents.
Capability Mechanics

- [{memory, disallow, {addr, 16#1500000, 16#1600000}}]
- [{memory, allow, {addr, 16#100000, 16#1600000}}, {memory, disallow, all}]
- [{memory, disallow, all}, {memory, allow, {addr, 16#1600000, 16#6400000}}]
Trees of Decreasingly Capable Processes

- **'Sandboxed' Console**
  - \{memory, disallow, all\}, \{ports, disallow, all\}

- **Keyboard Driver**
  - \{ports, allow, [16#60, 16#64]\}

- **Malicious App**
  - \{memory, disallow, all\}, \{ports, disallow, all\}

- **Limited Worker Process**
  - \{memory, disallow, all\}, \{ports, disallow, all\}
  - \{memory, allow, \{addr, 16#100000, 16#1500000\}\}

Sam Williams (University of Kent) How to Build an OS in Erlang: A Whistle-stop Tour of HydrOS
June 9, 2017 31 / 34
Demo!

- Basic terminal usage.
- Window and system management.
- Killing and restarting a live HydrOS node.
Acknowledgements

Special thanks for help with this presentation:

- Simon Thompson, University of Kent.
- Maxim Kharchenko, Erlang on Xen.
- Neeraj Sharma, Erlang on Rumprun

HydrOS Contributors:

- George Bates
- James Forward
- Anton Thomasson

Thanks also go to the ESPRC for funding the research.
Try it out!

Prebuilt images, sources, and build instructions are available at http://hydros-project.org.

I can be contacted at secw2 [at] kent [dot] ac [dot] uk.