Machine View

An OS that survives catastrophic failure

Machine View

Sam Williams, University of Kent

Machine View

HydrOS

An operating system written in Erlang, using the BEAM as a kernel.

- Built to withstand both software and hardware failures.

- One BEAM per core, with memory distributed between the cores.

- All operating system services and hosted programs are written in Erlang.

- Built using a multikernel design.

Machine View

Talk Outline

- What is HydrOS?
- What are multikernels?
- How does it work?
- OS Security
- Multiunikernels
- Future work

Machine View

Project Aims

Design and implement an operating system that is not only resilient to software faults, but hardware faults too. This includes continuing some program execution through:

- CPU core failure, or in multi-socket machines, total CPU failure.

- RAM failure

- Miscellaneous hardware failure (PCI devices, HDDs, DVD drives, etc).

- Resistance to catastrophic OS subsystem failure.

Machine View

Potential Usage Environments

- Systems that need to run for longer than the expected lifespan of components of the host machine.

- Places where systems cannot be reached to be repaired by humans.

- Arenas where physical damage to the hardware of the machine is likely.

Machine View

Multikernels

A design concept heralded by the Barrelfish and Factored OS projects in the late 2000s.

- Modern computers are analogous to distributed computing environments.

- The architecture of our current OSs will not efficiently utilise many-core computers.

- By treating OS design as a distributed problem we can create OSs that have better scaling properties.

Machine View

Multikernels In Practise

- Treats each core in a machine as if it were its own computer.

- Provide separated memory areas for each core.

- Message passing, rather than shared memory, used to co-ordinate processes.

- Use solutions from the distributed systems world to solve OS scaling problems.

Machine View

Project state

A working prototype operating system that is resilient to complete software node failure within a machine.

- The OS can run arbitrary Erlang programs, library availability depending.
- Support for more than 17 devices (to varying degrees of completeness), all directly written in Erlang.
- A multiunikernel framework that allows hosting of C programs on their own private nodes.
- A coarse-grained capabilities system that allows for in-depth control over the execute of programs.
- Work towards categorising and uncovering failure states within Intel, x86 machines.

Machine View

System Design

A network of communicating BEAMs within one machine.

- One Erlang node per core.

- Inter-core communication achieved through message passing only, no shared memory.

- All program code is written in Erlang and executed in a managed environment.

Machine View

Why Erlang?

The central HydrOS design principle is fault tolerance through concurrency.

- Erlang provides us with a language with strong support for concurrency and associated tools.

- Erlang's message passing system matches well with the multikernel inter-process communication paradigm.

- In many ways, the BEAM already resembles an OS.

Machine View

How does it work?

Machine View

System Components

- Bespoke bootloader.

- Minimal C standard library, tailored to the BEAM.

- The BEAM, with minor modifications.

- A small BIF library that provides data access primitives (port IO, memory mapped register access, etc).

- OS subsystems, written completely in Erlang.

- A set of drivers for basic devices found in most machines.

Machine View

Bootloader

A novel bootloader design that 'co-operatively' boots the system.

All BIOSes come with support to read from the provided boot medium while in 16 bit real mode.
These routines are not available in 32 or 64 bit mode.
Access to memory above 2mb is not possible in 16 bit mode.

Reading from USB (and others) devices at boot time is complicated and requires interaction with a large number of system components.

Subsequently, we utilise the multicore nature of modern systems to avoid having to write large amounts of C driver code (that would later have to be duplicated in Erlang).

Machine View

Bootloader (2)

We do this by booting the first core in the machine into 64 bit mode, then starting another core in the system, which stays in 16 bit mode.

The second core (the 'real mode slave') uses BIOS routines to read chunks from the boot device into low memory.

The primary core then picks up these chunks and moves them into high memory.

Machine View

C Standard Library

- Created to be the minimal required to compile and run the BEAM.

- Headers required for BEAM compilation taken from GNU Lib C.

- 154 function definition stubs.

- 24 implemented functions.

- 83kb compiled.

Machine View

The BEAM

- Source mostly unmodified, providing good compatibility with normal, OS hosted Erlang systems.

- Most of the deviations from a normal BEAM are achieved through options in the build config file.

- Only a dozen or so edits to the BEAM are required to run in our environment.

- A number of files have been removed from the source tree that were not used in the output system.

Machine View

OS Services

All written completely in Erlang, communicating with hardware through a small library of BIFs, where necessary.

—

Provides systems for handling:

- Interprocessor communication

- Machine topology and OS state

- Device driver management

- Monitoring and restarting of external nodes

- A concurrent init system

- A basic Erlang shell

Machine View

Drivers

The OS provides support for a number of basic devices, all written in Erlang.

- Various timers
- Local and IO APICs
- Keyboard and cursor control
- Various system tables (for example, SMBIOS, PCI configuration space, etc.)

Ethernet drivers are next on the list.

Machine View

Design Principles By System Layer

The kernel layer: The BEAM, interrupt handlers, IO BIF library.
- Kept to a complete minimal. Whenever possible, systems are pushed to the next layer.
- Never communicates directly with other cores.
- Code kept short and concise, whenever possible.

The 'local' Erlang OS layer: Services that must be duplicated on each node.
- As there is no SMP at the BEAM level, services that require low latencies must embrace shared memory (through ETS tables).
- Can communicate with their counterparts on other nodes.
- Certain BIF calls are intercepted at this level and rerouted to HydrOS specific Erlang implementations.

Machine View

Design Principles By System Layer (2)

The 'global' Erlang OS layer: Programs that present a single service, the processes of which may be distributed over many nodes.

- No shared memory, communication only achieved through message passing.
- Process node placement determined by the OS spawn system.

Machine View

Security Through Managed Execution

HydrOS Erlang programs are written exclusively in Erlang.

Because instructions are interpreted by the BEAM during execution, we have much more control over program execution than with traditional machine code programs.

This allows us to do away with traditional, hardware based permission rings and controls, opting for much more precise, fine-grained, software based controls.

Machine View

Capabilities

Capabilities provide the OS user with a rich language for limiting user program activities.

The capabilities of a process are essentially defined as an exclusion list.

- For example, '[]' specifies that there are no limits to a processes behaviour, whereas the list '[{raw_memory, 0, 16#200000}, {port_io, all}]' specifies that the process cannot access IO ports or the first 2mb of virtual memory.

The root process in the system is started with no capability exclusions.

Machine View

Capabilities (2)

When processes are spawned capability exclusions can be specified, but by default they are given the same capability exclusions as their parents.
A process cannot remove exclusions from its capabilities once it has spawned.

- This means that a system user can freely run untrusted code safely, by ensuring that it does not have dangerous capabilities.
- Similarly, damage from potentially haywire OS subsystems can be limited by spawning them with only the capabilities that they require.

Application authors can also extend the system by adding their own capabilities and appropriate checks to the system.

Machine View

Unikernels

Unikernels, a technology popular in virtualisation, are compiled library operating systems that run only a single application.

Unikernel systems take a single source program and turn it into a highly specified operating system image that is normally deployed to cloud environments.

This style of system has a number of advantages, including vastly lowering the startup time of new cloud instances, as well as lowering the size of the attack surface of applications running in the cloud.

The 'real mode slave' system mentioned earlier is essentially an example of a unikernel.

Machine View

Multiunikernels

HydrOS provides a framework for hosting unikernels within an Erlang based multikernel system.

- Normal BEAM kernels are replaced on specified nodes with unikernels that have been compiled with the provided toolkit.

- These unikernel nodes run alongside their Erlang counterparts, communicating across the same node mailboxes.

- Unikernel nodes in the system have complete control over the cores on which they run, leading to extremely fast execution of the hosted programs.

Machine View

Multiunikernels (2)

- This framework may be helpful in situations where part of a hosted application does not play to Erlang's strengths.

- Multiple unikernels can be hosted on the same machine, while being orchestrated by an Erlang node.

Machine View

What Next?

- Find ways of causing CPU core failure during execution that mimics sensible real world scenarios.

- RAM failure fault tolerance.

- Expanding the model out of the box, and onto the network.

- A distributed file system.

- A concurrent GUI.

Machine View

Try it out!

Prebuilt images of the system, as well as the source code, are available at http://hydros-project.org.

- The OS runs on Bochs, Qemu, KVM and Xen, as well as all real machines that we have tried it on. Your mileage may vary!
- Use the system at your own risk. It is 'seriously alpha' software. Any damage incurred upon your system is your responsibility!

The system can be compiled from source on any *nix like system, after having generated a simple cross-compiler with crosstool-ng.
Build and usage instructions are available on the website.

Have fun, and let us know what you think!

Machine View

Questions?