A microkernel written in Rust

Porting the UNIX-like Redox OS to Arm v8.0

Robin Randhawa

Arm

February 2019

I want to talk about

Redox

on

arm

Redox is written in Rust - a fairly new programming language

So it is important to discuss Rust too

My goals with this presentation are

To primarily talk about these

- Lightweight intro to the Rust language
- Unique features that make it shine

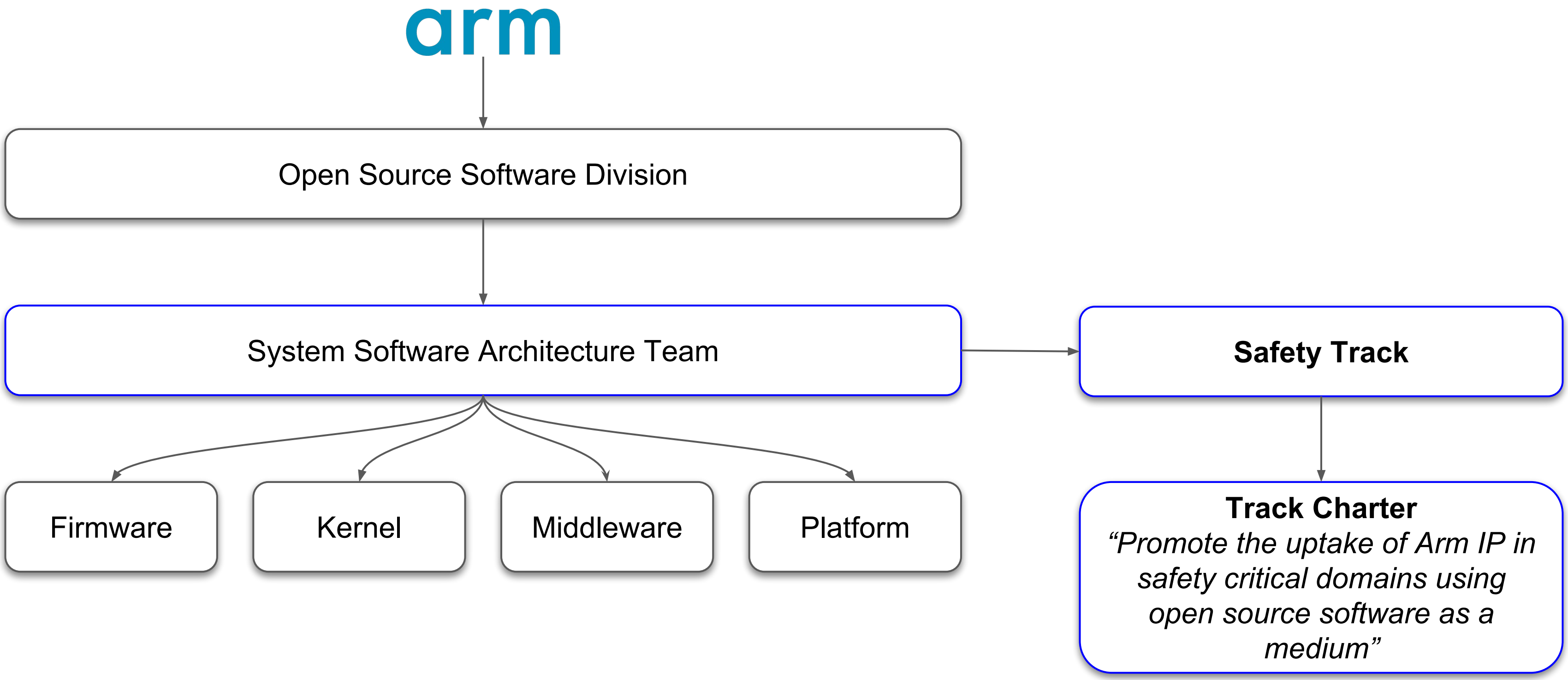
arm

- Explain why Rust is interesting for arm
- Rust's support for arm designs

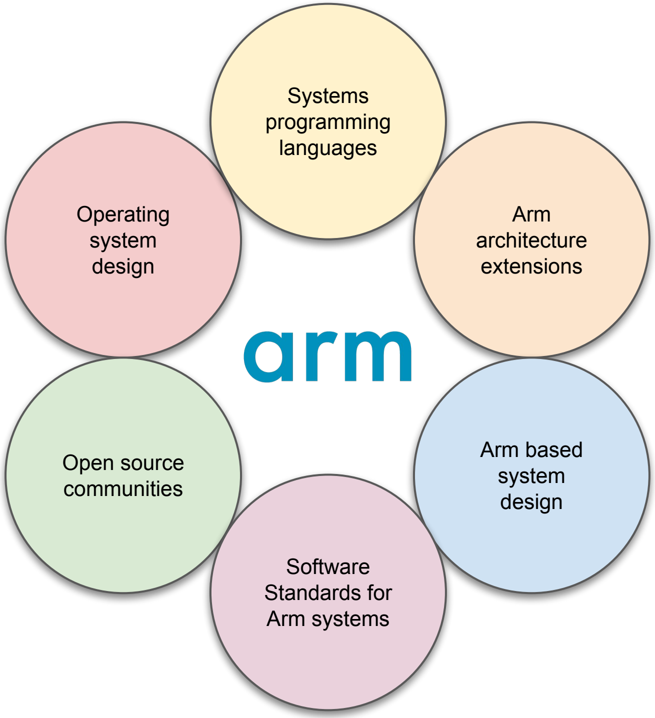
Redox

- Introduce Redox's history, design, community
- Status, plans

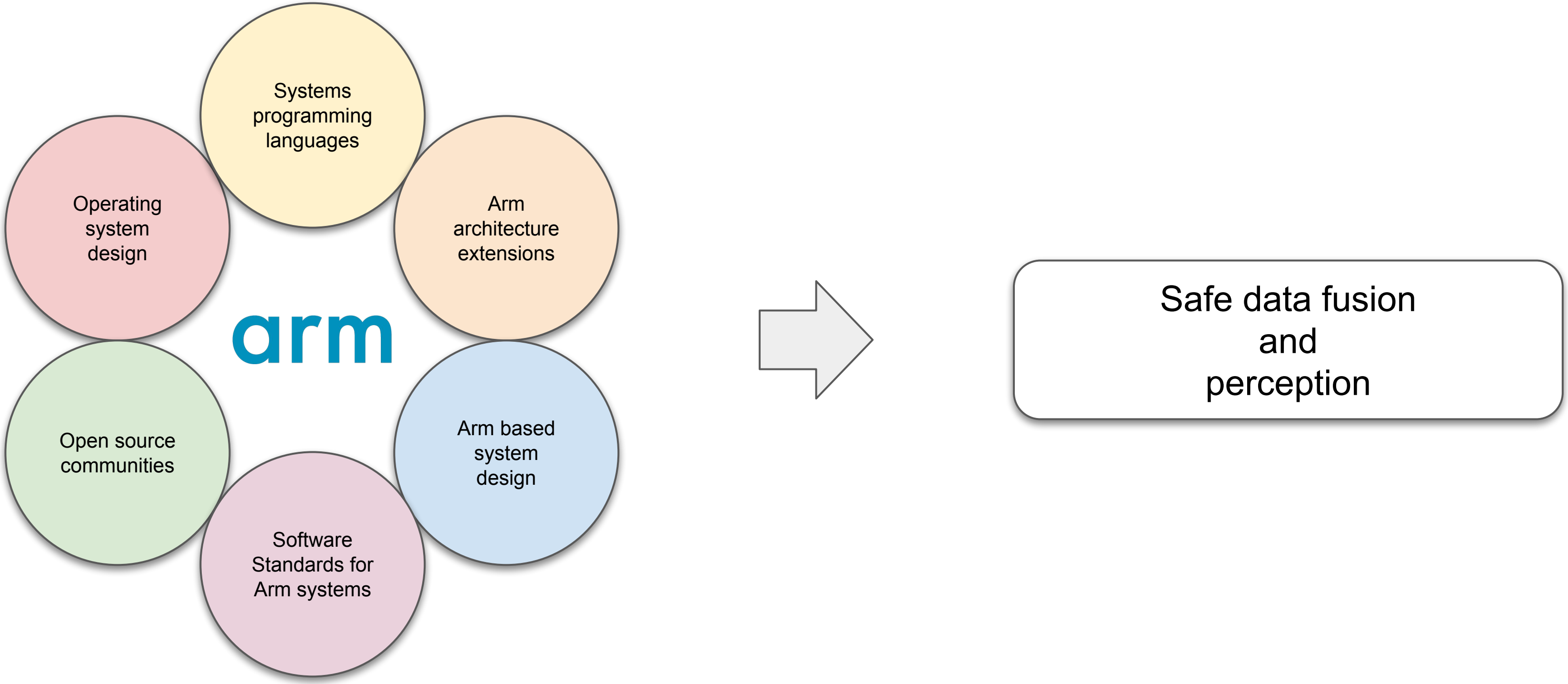
… and some relevant anecdotes from the industry


arm
Open Source Software Division
System Software Architecture Team
Safety Track
Firmware
Kernel
Middleware
Platform
Track Charter
“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

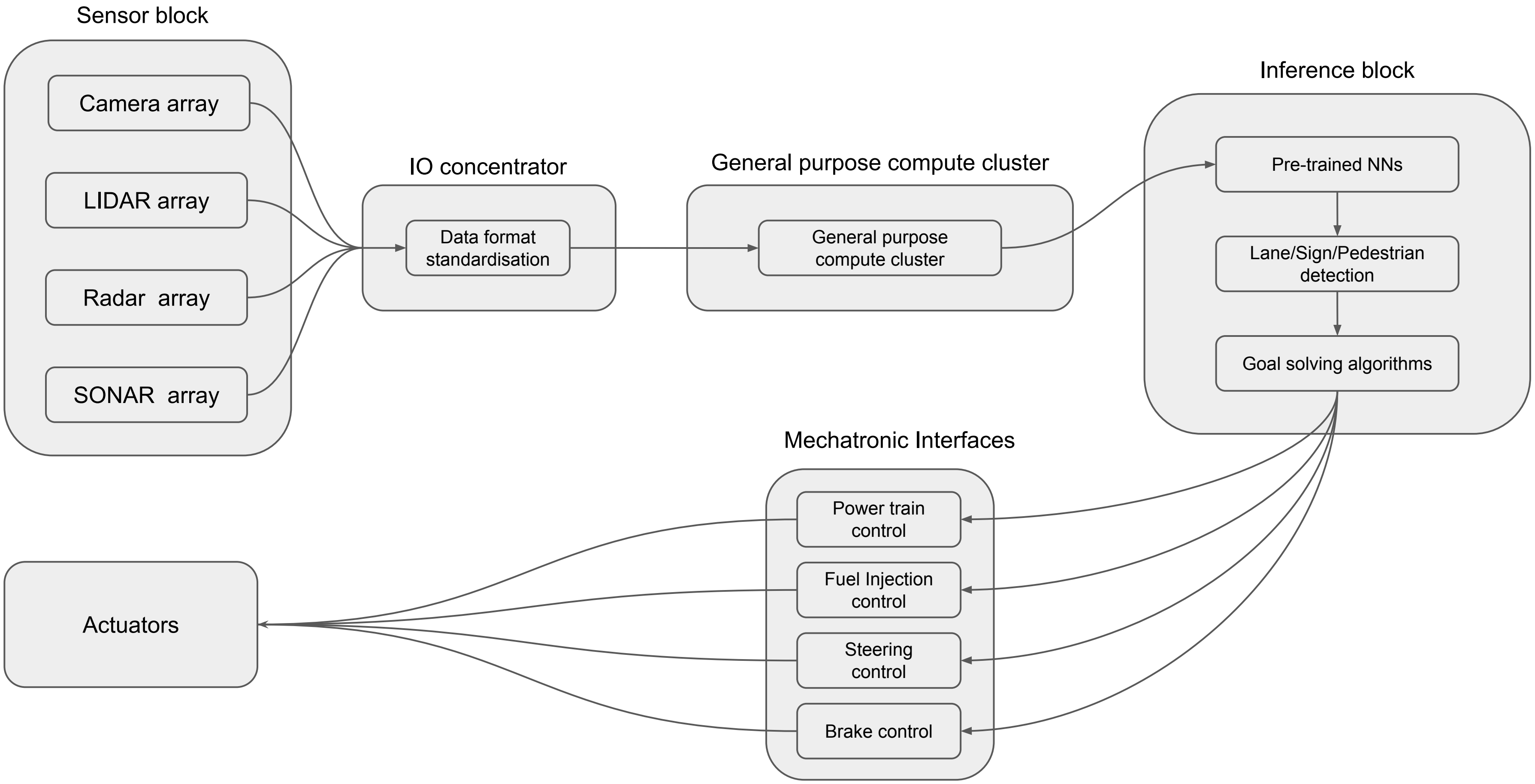
My areas of Interest



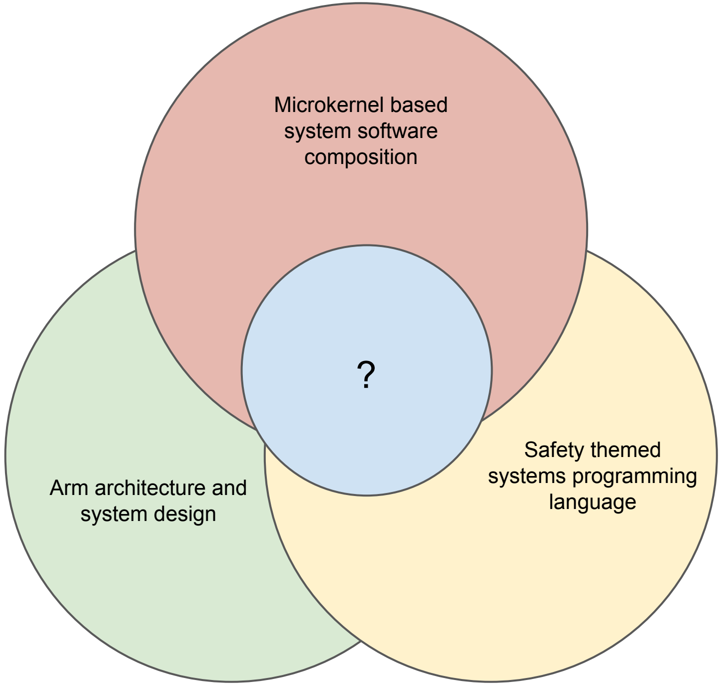
Primary focus area



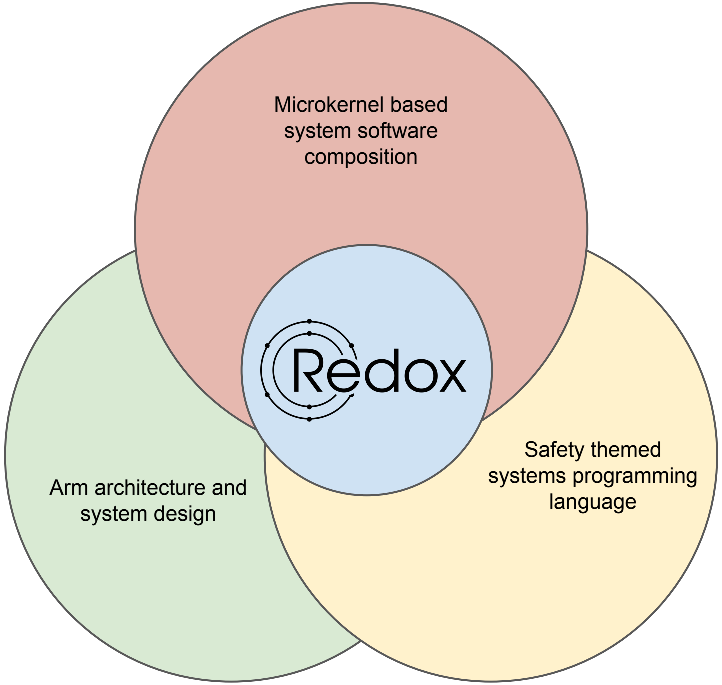
Data fusion and perception pipeline



My explorations needed something at this intersection

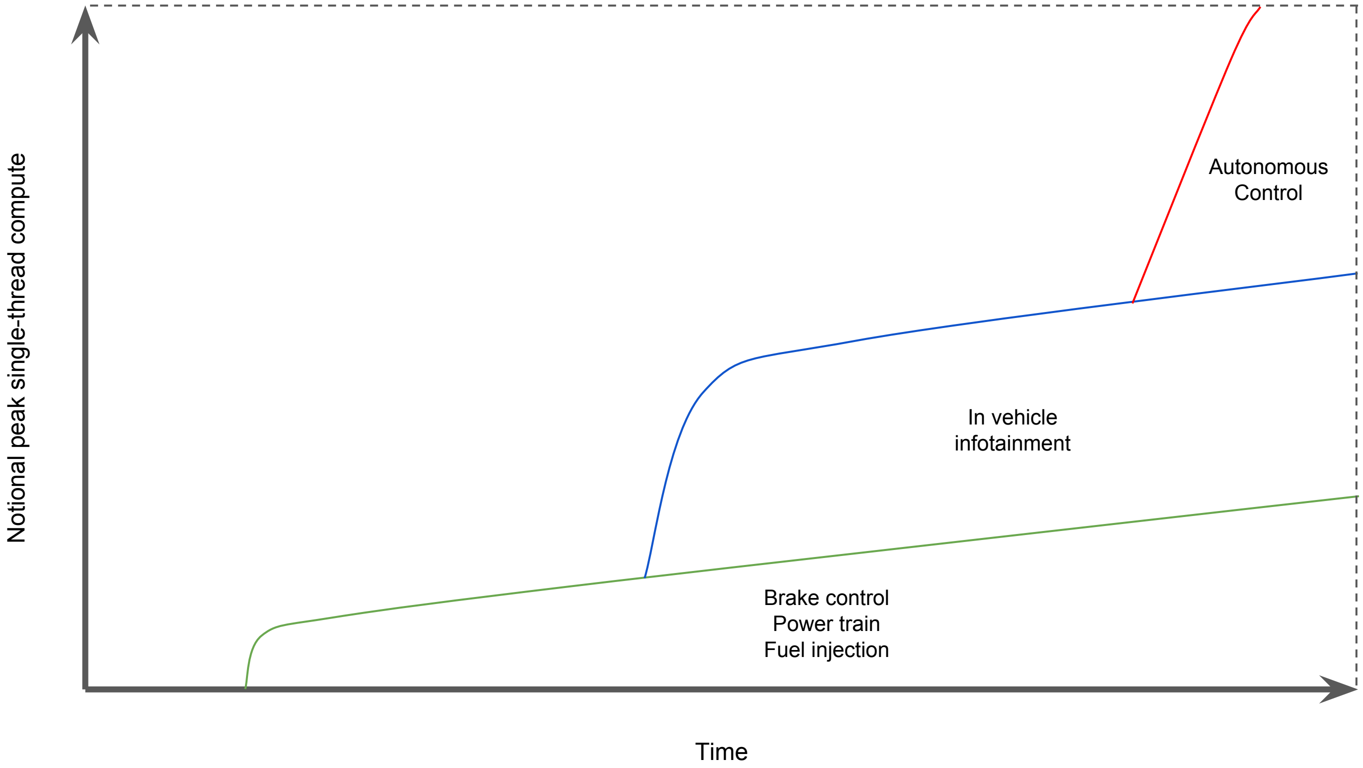


I started writing my own microkernel in Rust…. then chanced upon Redox OS

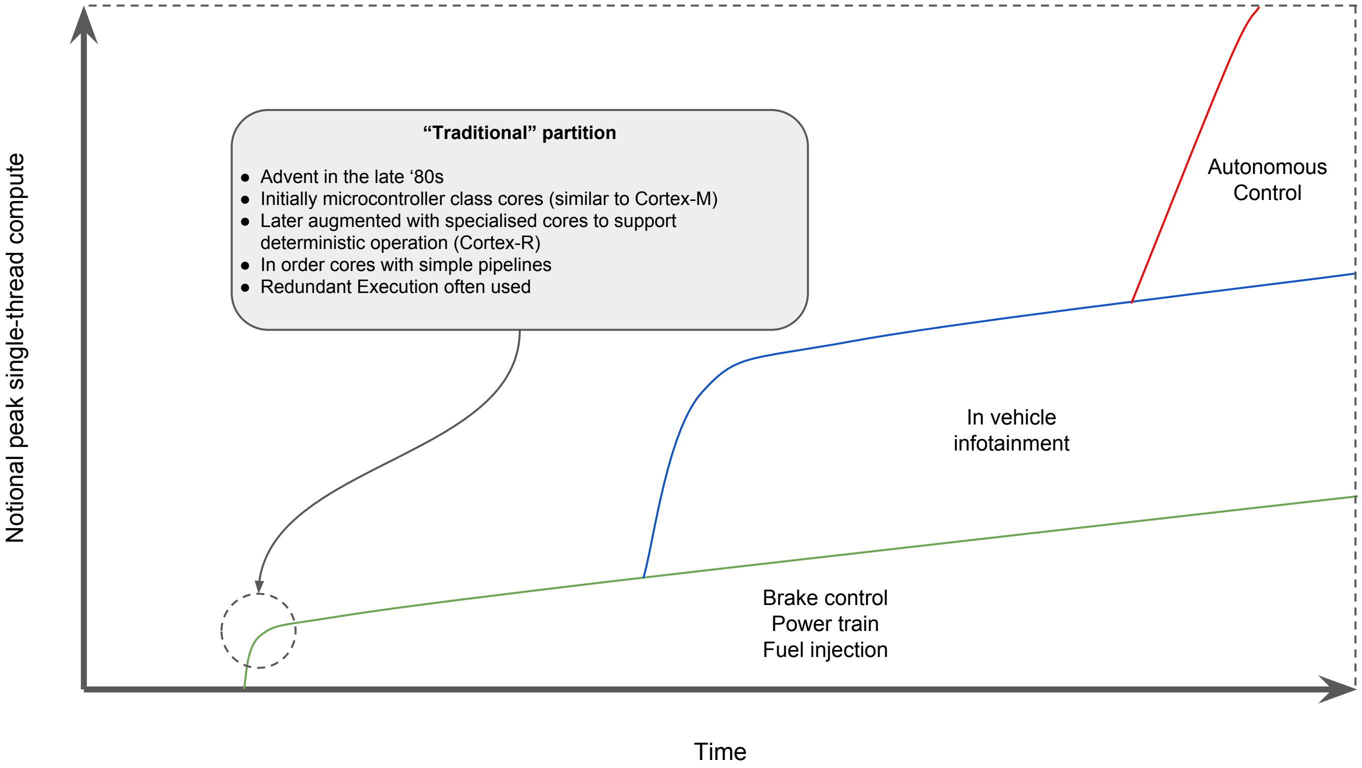


I see a worrying paradox in the making...

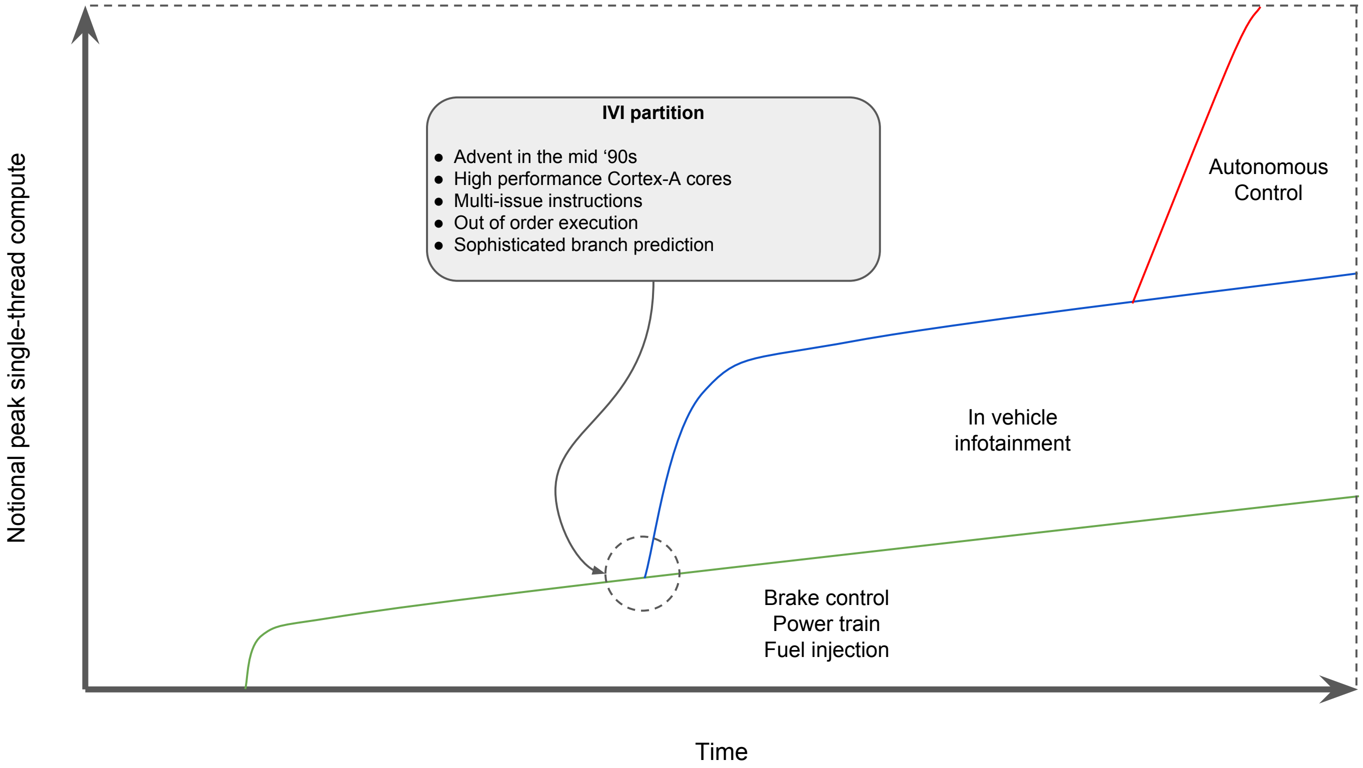
The compute requirement for automotive autonomous functions is insanely high



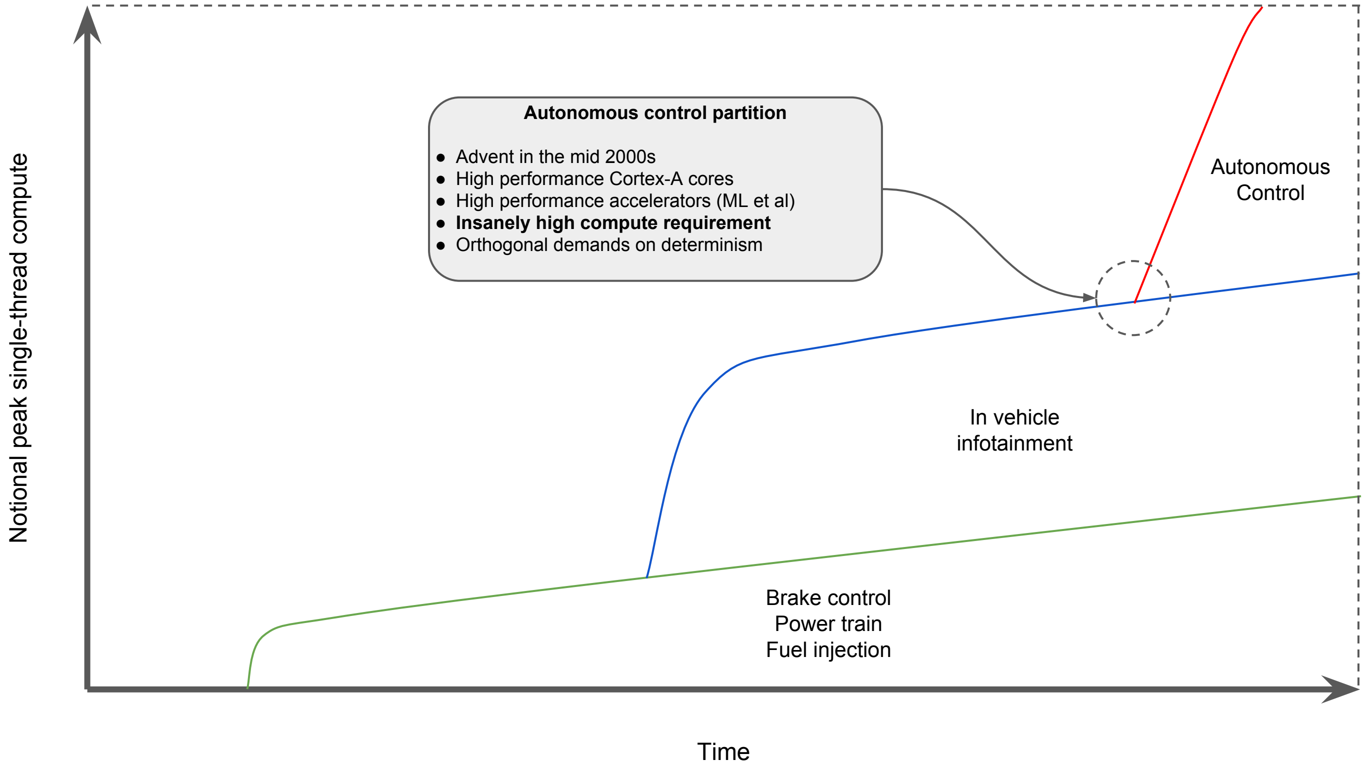
The compute requirement for automotive autonomous functions is insanely high



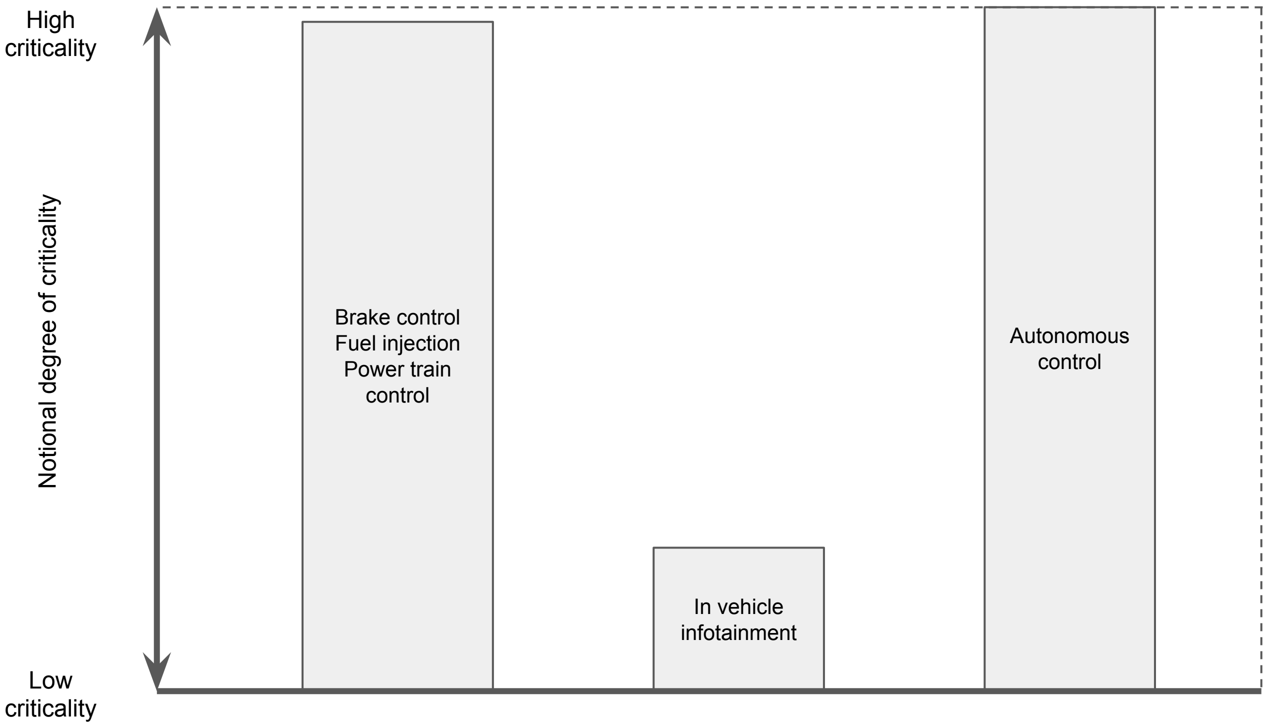
The compute requirement for automotive autonomous functions is insanely high



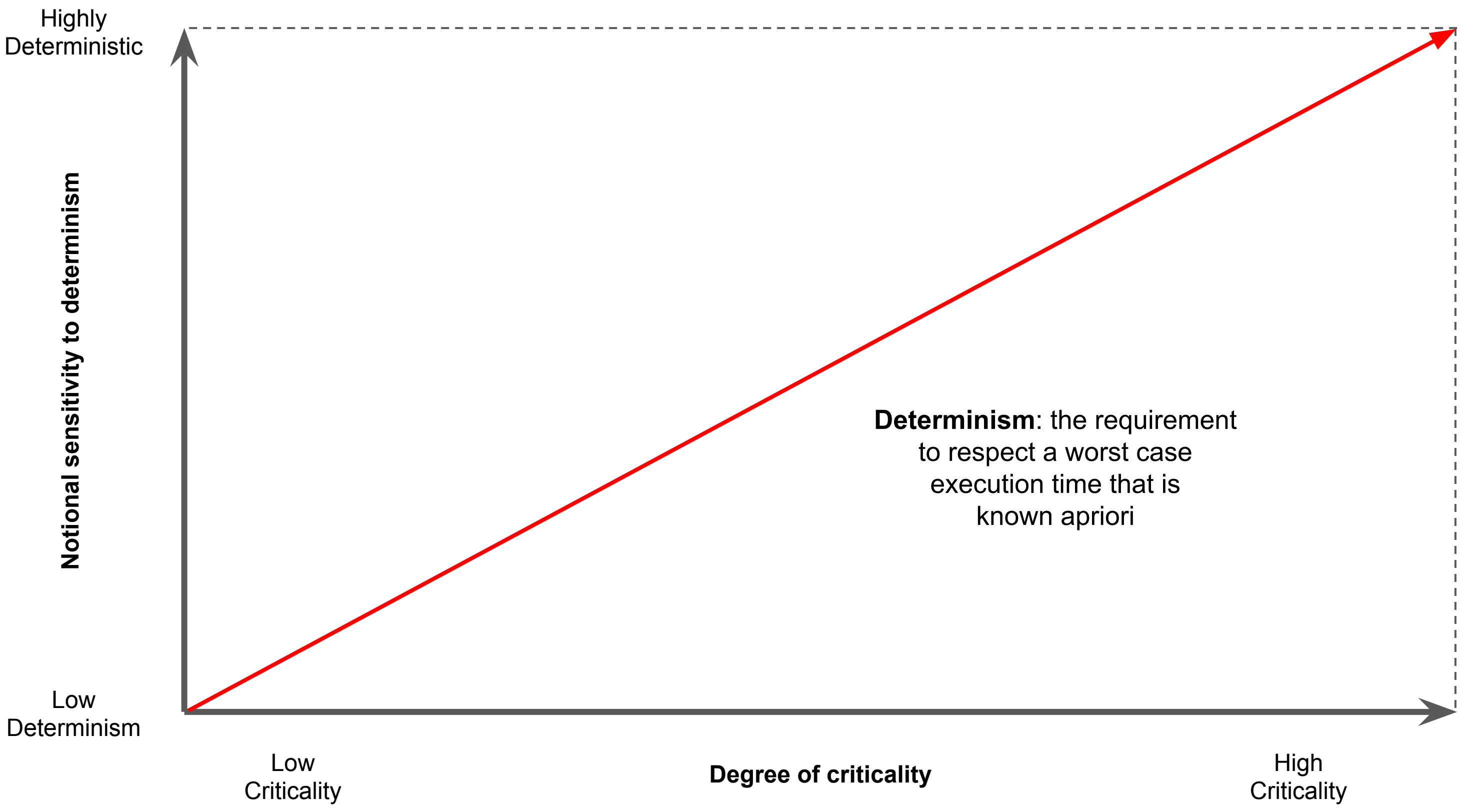
The compute requirement for automotive autonomous functions is insanely high



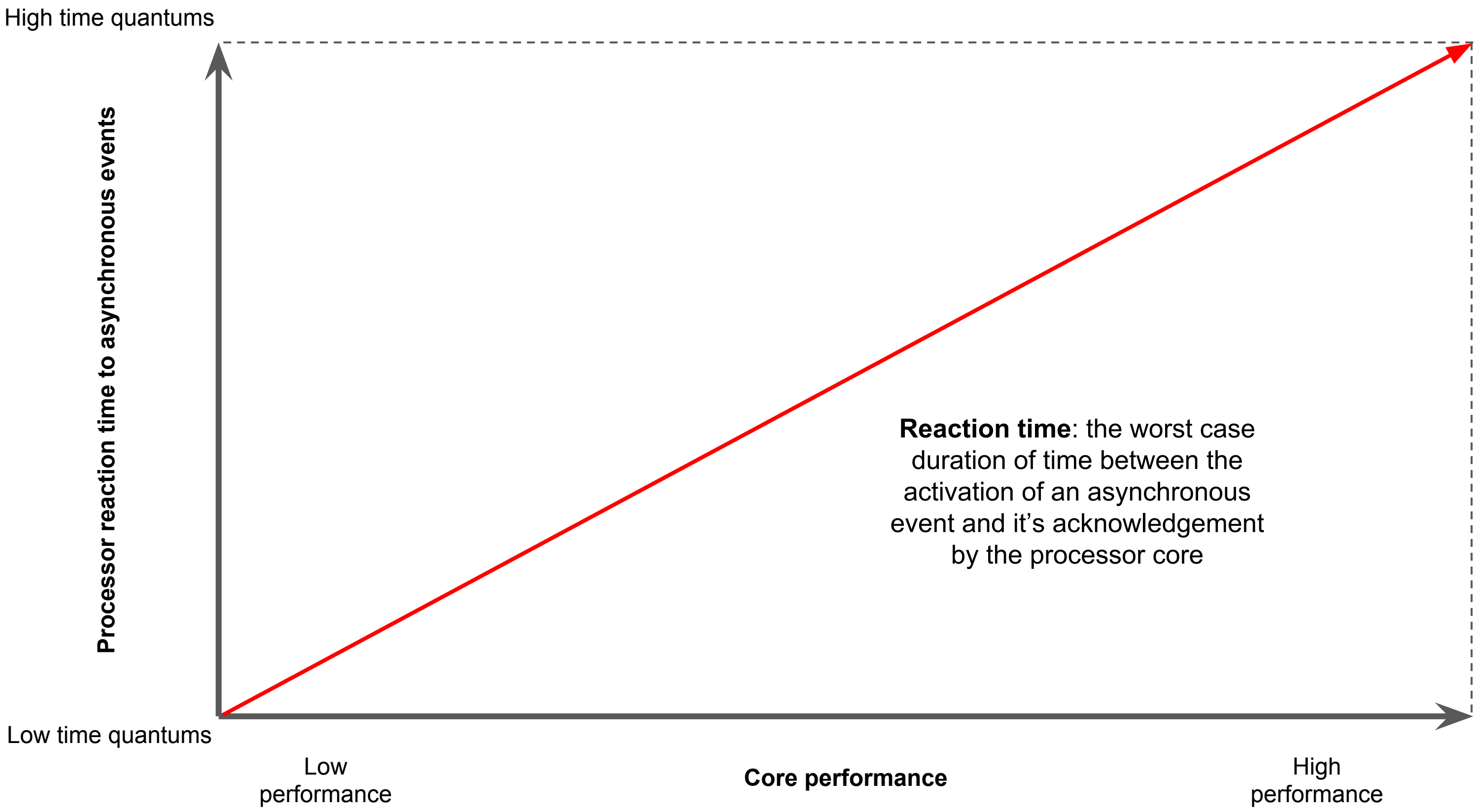
Autonomous control has very high criticality requirements



In general, the sensitivity to deterministic execution and the degree of criticality are linearly related



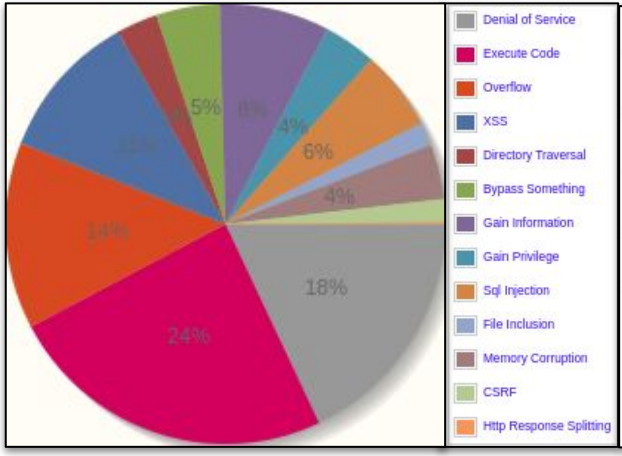
In general, a processor's performance and it's "reaction time" are linearly related

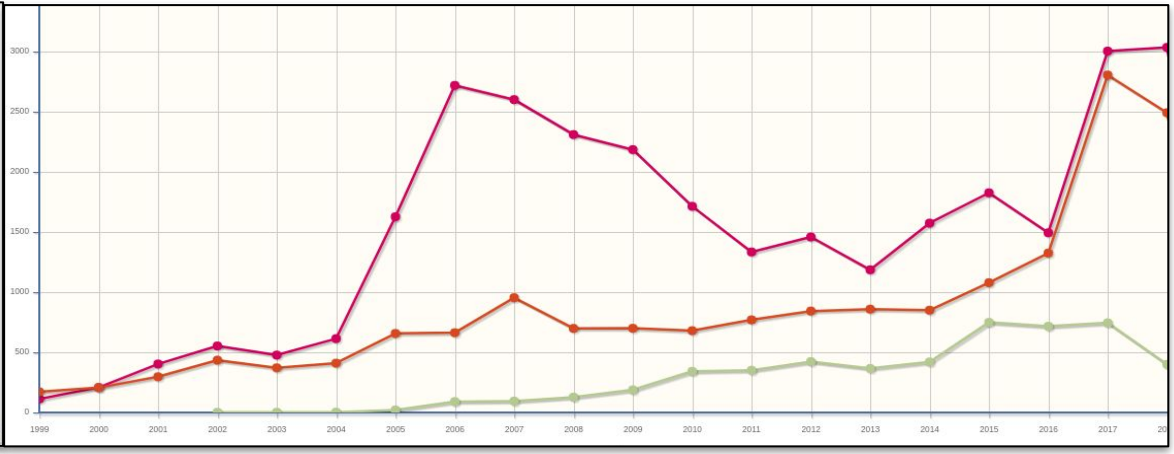


In summary...

- Autonomous control has very high criticality requirements
- Autonomous control has very high performance requirements
- High criticality requires very deterministic execution
- The higher the processor’s performance the slower it’s reaction time
- **Paradox: For autonomous functions, the required higher performance seemingly cannot be had deterministically and with low reaction times**

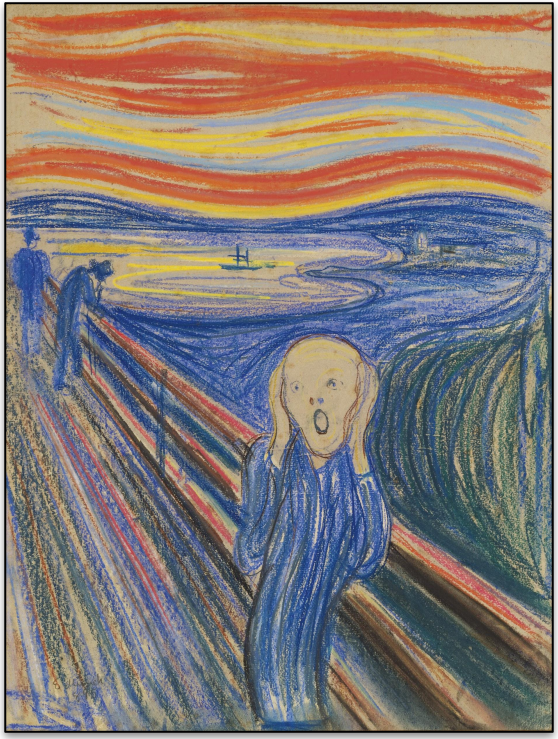
There is a thin line between safety and security





Complexity is on the rise...

Insanity



So...

Autonomous functions are becoming increasingly pervasive

Hardware engineers are working hard to make the hardware sensibly safe

Despite their best attempts, it is very likely that software for such systems will be exceedingly complex

Any and every attempt to make complex software safe is welcome

Traditional approaches to the problem

Mixed criticality hardware and software designs

Traditional quality management of hardware and software

Reliance on “safe dialects” of C (MISRA et al)

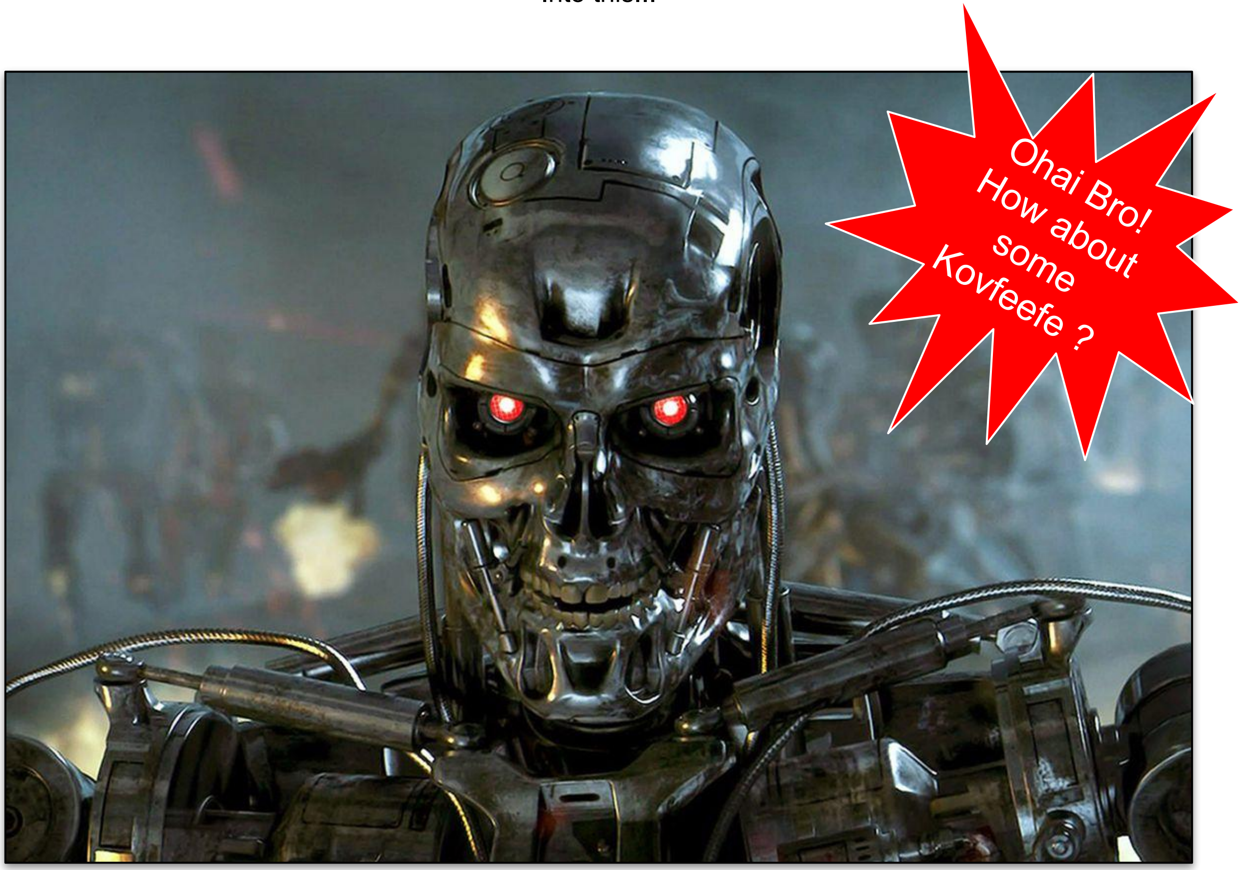
Formal verification of hardware and software

How about:

A language designed for safety that provides guarantees without compromising performance ?

We can't let this...

Into this...



https://www.rust-lang.org/

```
fn main() {
    println!("Hello, world!");
}
```

"Rust is like doing parkour while suspended on strings & wearing protective gear.

Yes, it will sometimes look a little ridiculous, but you'll be able to do all sorts of cool moves without hurting yourself."

- Snippet from Reddit conversation about Rust

"It wasn't always so clear, but the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before."

- The Rust Book Introduction

(https://doc.rust-lang.org/book/)

“Rust is very expressive”

“I often use Rust instead of Python or Ruby”

- Me

```
use std::process::Command;

Command::new("ls")
        .arg("-l")
        .arg("-a")
        .spawn()
        .expect("ls command failed to start");
```

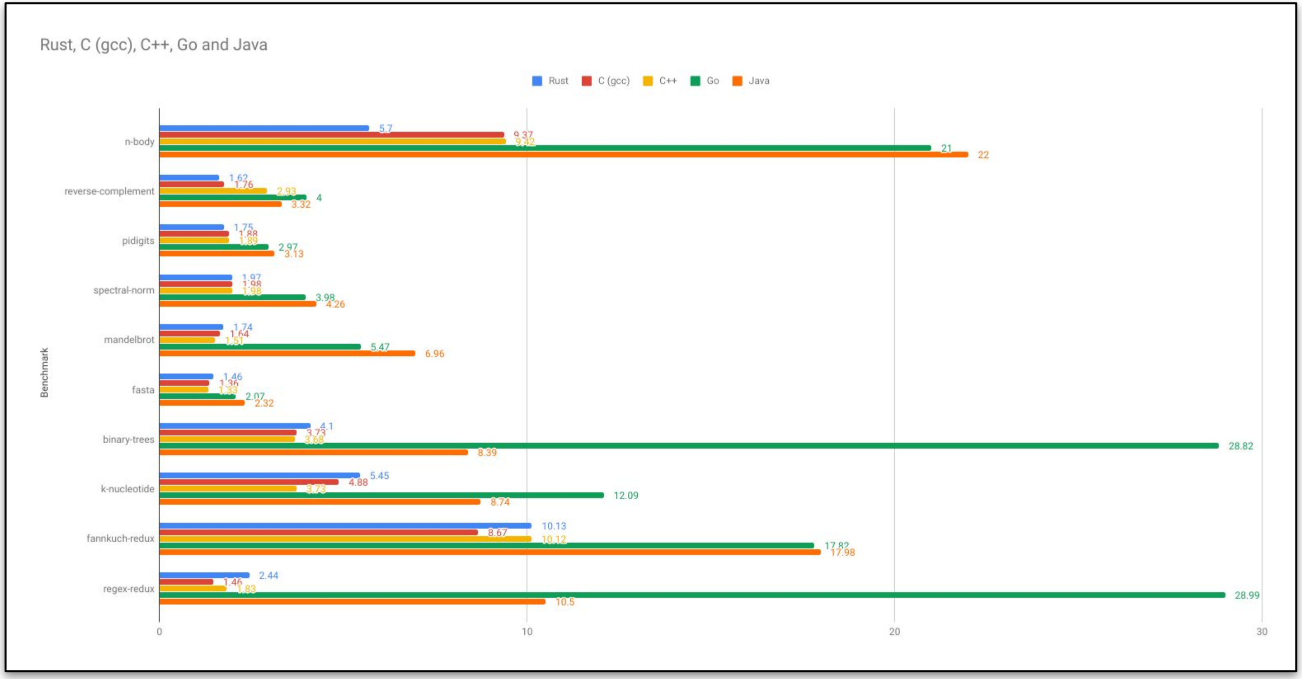
"Rust's expressiveness is great for making complex systems software concepts accessible"

- Me (again)

```
/// Map a page to a frame
pub fn map_to(&mut self, page: Page, frame: Frame, flags: EntryFlags) -> MapperFlush {
    let p3 = self.p4_mut().next_table_create(page.p4_index());
    let p2 = p3.next_table_create(page.p3_index());
    let p1 = p2.next_table_create(page.p2_index());

    p1[page.p1_index()].set(frame, flags | EntryFlags::PRESENT);
    MapperFlush::new(page)
}
```

"The performance of machine code generated from idiomatic Rust is typically at par or better than machine code generated from idiomatic C++"



amazon

Google

Dropbox

ATLASSIAN

unity

Microsoft

- With Rust
  - You can’t forget to explicitly initialise variables
  - You can’t overflow an array
  - You can’t forget to free memory allocated on the heap
  - If shared data is protected by a lock, you cannot forget to take the lock first
  - You cannot have a dangling pointer
  - A double free of memory is not possible
  - Use after free of memory is not possible
  - Generally speaking there is no undefined behaviour

.. and this is all checked at compile time for you

- Rust is actually a combination of 2 languages: Safe Rust and Unsafe Rust

    - Safe Rust
        - Is the default
        - Using it will ensure that you have no type safety or memory safety issues
        - Even for concurrently executing code
        - The compiler checks this for you
        - Clever static analysis ensures there is no performance hit
        - Code generated from idiomatic Safe Rust is typically better performing or at par to Code generated from idiomatic C, C++
        - Safe Rust limits the programmer from using “raw” pointers

    - Unsafe Rust
        - Is enabled by explicitly annotating code as unsafe
        - Disables the comprehensive compiler checks to permit C/C++ like type and memory operation
        - Code generated from unsafe Rust is typically at par with C and C++

- Basically, Rust enables the programmer to opt out of it’s strict safety rules if desired
- Annotating unsafe code means that if there is a failure, you know exactly where the problem is - unlike C and C++ where for similar situations you may not be able to tell easily

- Rust is
  - Not an interpreted language
    - Rust code is compiled to native machine code
  - Has no garbage collector and none of the associated non-determinism
    - Instead, rust's rules ensure correct alloc/dealloc of memory including across concurrent contexts***: all checked at compile time!***
  - Is a statically typed language
    - The compiler requires the types of all variables to be known at compile time
    - But the compiler is smart and can infer types itself many cases
  - Before compilation succeeds, Rust requires the programmer to:
    - Acknowledge any possibility of error
    - Take some suitable action

  This is unlike most languages that put the onus for error checking on the programmers…. Who are lazy….

- Rust doesn’t have any exception handling!
  - Instead Rust groups errors into *recoverable* and *non-recoverable* error types
  - For managing recoverable errors Rust provides a special type: `Result<T,E>`
    - This type enables intuitive error introspection without the possibility of neglecting any outcome
  - For unrecoverable errors, Rust has the `panic!` Macro
    - The macro enables consistent responses to such errors without any ambiguous side effects

- Data is immutable by default in Rust
  - Simple idea - shaves off a significant set of memory safety problems
  - If data is immutable by default - you can't change it unless you first declare it as mutable

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}

error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

- Rust has no numerical type-width ambiguity
  - Unlike C and C++, Rust's types encode the type-width in the type names
    - Unsigned integers
      - u8 u16 u32 u64 u128
      - usize (machine word size)
    - Signed integers
      - i8 i16 i32 i64 i128
      - isize (machine word size)
    - Floats
      - f32 f64
- Rust is generally better defined and not ambiguous as other systems languages like C, C++

- Rust doesn't have C++ like classes
  - Rust has C-like structs for creating programmer defined composite types

```
struct Record {
   id: u32,
   data: Vec<u32>,
}
```

  - Structs have functions associated with them that enable the expression of type specific behaviours
  - Behaviours can be specified across types using the concept of Traits
    - Traits express an interface each type is required to have
  - Rust is like C++ but without the baggage of Classes, multiple inheritance complexity etc

- Rust has generics
  - For types, methods and more

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

- Traits express desired behaviours from types
- Including abstract generic types
- “Trait Bounds” allow functions to place compile time restrictions on type arguments

- Rust has Atomics
    - With support for expressing the desired memory consistency when working with Atomic types
        - Relaxed, Release, Acquire, AcqRel, SeqCst
    - Memory consistency semantics follow LLVM’s model (C11)
    - Easy to implement common synchronisation primitives using these Atomic types and Rust’s automatic reference counting mechanisms

- Ownership
  - Rust requires that every data item have an associated owner (variable)
  - When data is passed around, the ownership changes
  - Once ownership has changed attempting access to the data is prevented at compile time

- Borrowing
  - But passing data around implies expensive copying (for anything but trivial types)
  - Rust permits sharing data using the concept of borrowing references to the data
  - Just like other types, references are immutable by default
  - Rust explicitly checks that
    - There is only every 1 mutable reference to a given data item across all scoped
    - Multiple immutable references are permitted
    - Mutable and immutable references cannot mix

- Rust has excellent support for Threads

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

- Rust has a very rich standard library
  - Large collection of optimised modules
  - Vectors, Strings, Hashes maps etc
- Rust has super useful functional patterns
  - Iterators, generators, closures
- Rust has built-in support for test expression
  - With tooling to run and benchmark tests
- Rust supports generating documentation from code
  - Modern tooling that autogenerates HTML etc
- Rust has very good foriegn function interfacing capability
  - Call Rust code from other languages
  - Call other languages from Rust

- Tools
  - Rustup
    - Painless rust toolchain installation/maintenance/update
    - Painless toolchain target architecture switching
  - Cargo
    - Rust package manager
    - Like Ruby’s gems or Python’s pypi but way better
    - Cargo packages are called ‘crates’
    - Cargo uses semantic versioning for crates for guaranteed dependency fingerprinting and replication
    - Cargo works with the crates.io central package repository
    - Seamless recompilation of crates to compiler supported toolchain targets

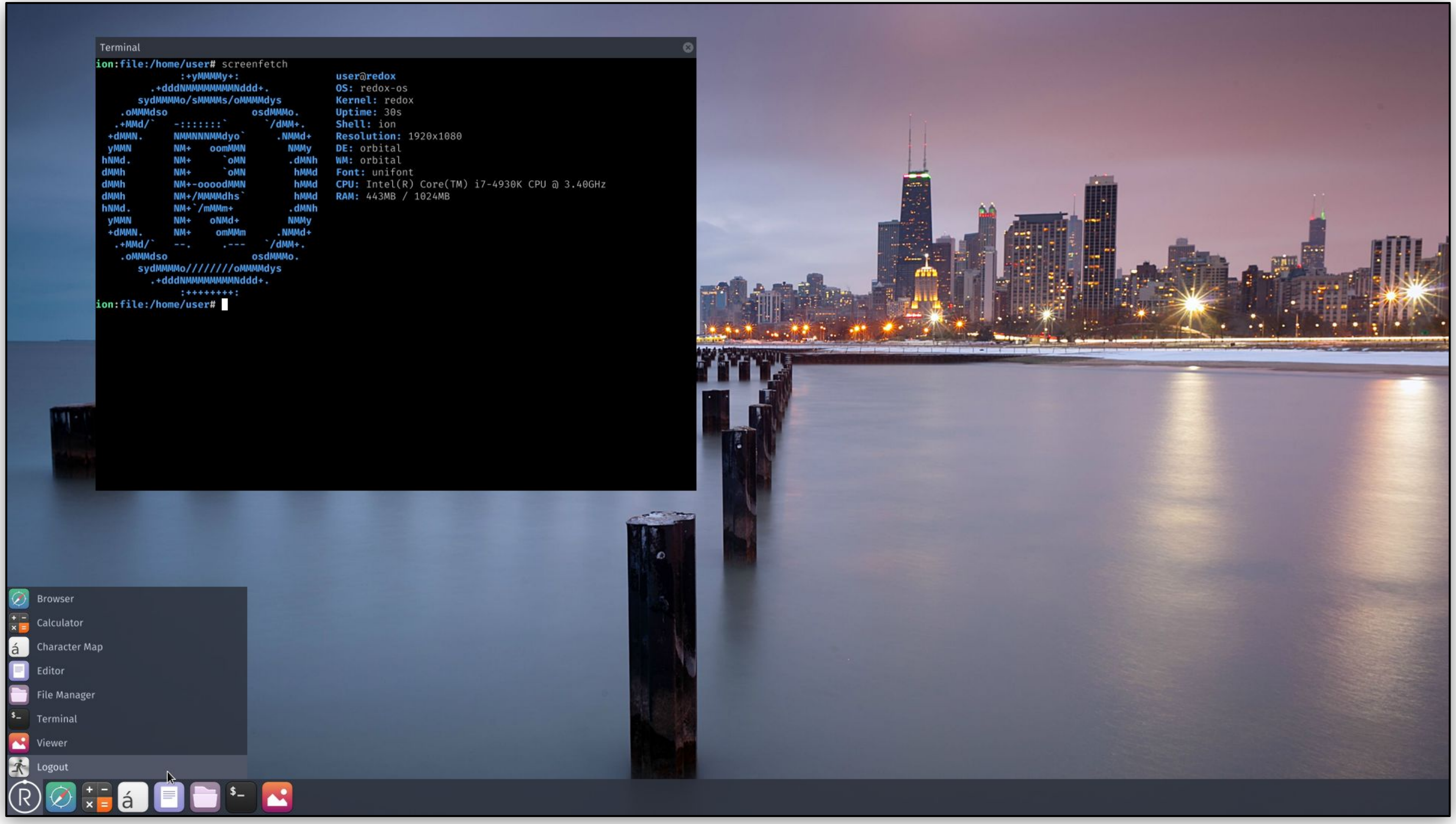
- My Rust ramp up sequence
  - The Rust Book
  - Rust by Example
  - The Rust Nomicon
  - The Rust Reference

- Was Rust genuinely useful for implementing a microkernel ?
    - Yes
    - unsafe Rust made it very easy for me to locate and root out correctness problems
    - The expressive nature of the language made it a pleasure to design and implement MMU abstractions
    - Interop with asm code was a breeze - the #[naked] decorator was useful
    - Writing synchronization code with abstract memory model expectations in Rust without needeing asm code was neat
    - The module subsystem was particularly useful

- What next for Rust and Arm ?
    - The Cortex-A embedded Working Group
    - The Cortex-M embedded Working Group
    - The Rust language specification Working Group (doesn't exist yet)
    - The RustBelt project

Redox

https://www.redox-os.org/


Terminal
ion:file:/home/user# screenfetch
user@redox
OS: redox-os
Kernel: redox
Uptime: 30s
Shell: ion
Resolution: 1920x1080
DE: orbital
WM: orbital
Font: unifont
CPU: Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz
RAM: 443MB / 1024MB
ion:file:/home/user#
Browser
Calculator
Character Map
Editor
File Manager
Terminal
Viewer
Logout

- An MIT licensed UNIX-like OS stack written in Rust
- With a Rust microkernel at its core
- Implements a reduced set of UNIX system calls
- Re-implements most UNIX components in Rust
- Provides a POSIX compliant C library - also written in Rust

- Rust (the chemical process) involves oxidation
- Redox (the chemical process) includes oxidation
- Redox sounds like UNIX (kind of)
- Rolls off the tongue easily!

- Leverage Rust
  - Showcase safe and secure software development using Rust
  - Use idiomatic Rust to make complex system software internals accessible to the lay programmer
- Leverage existing software
  - Enable easily re-building applications for existing UNIXen to run under Redox
- Cover a wide range of target domains
  - The primary focus has been the desktop domain
  - The currently emerging focus is the embedded domain
  - Long term goal is to target servers

- Written by Jeremy Soller (System76)
  - Initially tinkered with x86_64 assembly to “learn how computers work”
  - Was aiming to write a simple context switching mini-kernel in assembly for his PC
  - Had many headaches as a result but learnt a lot about pitfalls in low level OS design
  - Discovered Rust and found that Rust’s feature set was an excellent fit for safe, low level programming
  - Wrote incrementally complex bits using Rust: a simple bootloader, a mini graphics stack, an IO stack for mice and keyboards, a task scheduler
  - Got to a desktop environment and shared on github
- Then in 2015, someone told Reddit

- Steady development since then
  - EFI OS loader
  - C library
  - Pthreads support
  - RedoxFS file system
  - Driver library
  - Growing list of ported applications
- Google Summer of Code 2017
  - Made Redox self hosting
- Redox Summer of Code 2018
  - Added support for booting from ext2 filesystems
  - Began work on porting to Arm

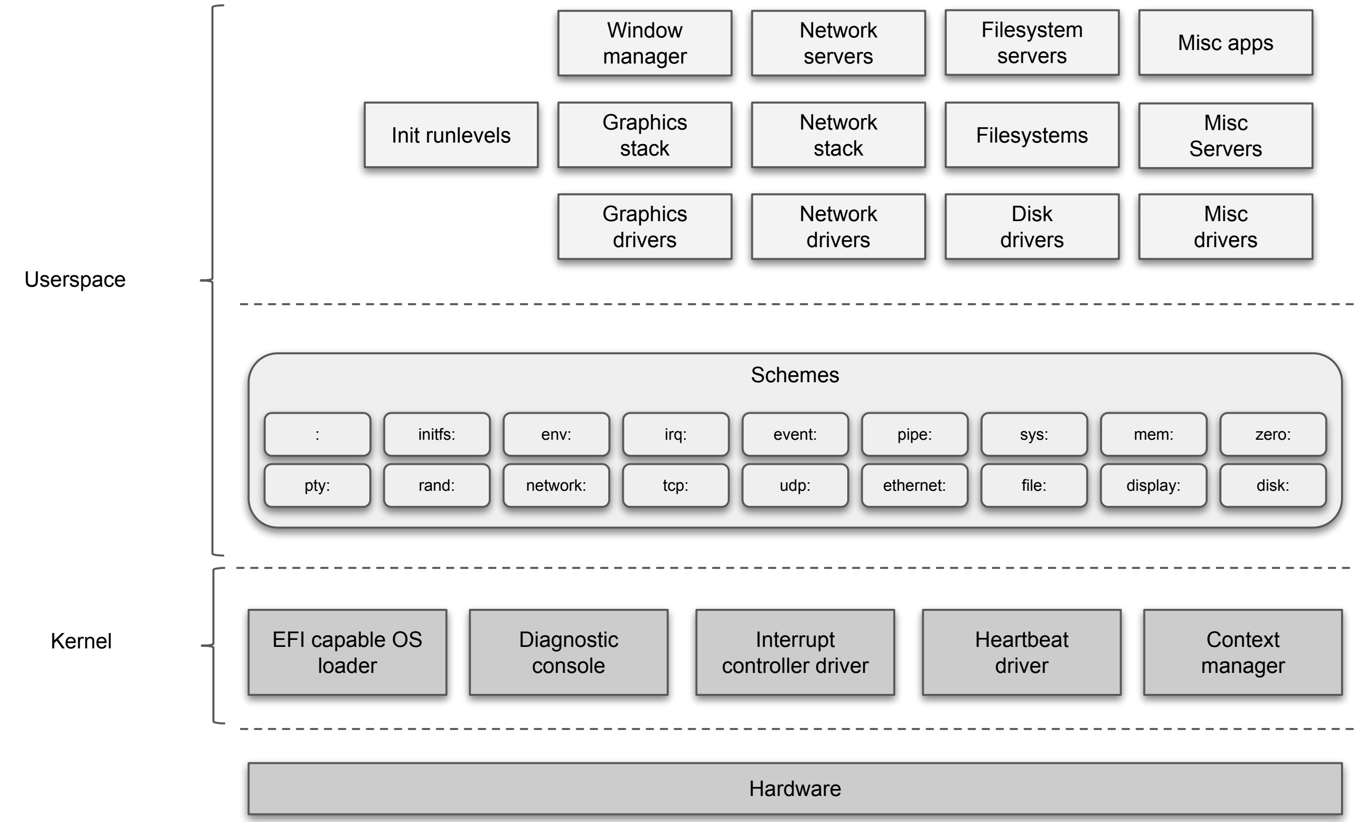
- Apps, libs

```
[~/work/repos/redox-workspace/redox.new/cookbook/recipes] [cookbook::aarch64-prep] [last: 0s]
robin@vulcan $ ls
acid             coreutils     ffmpeg          gigalomania    ipcd          lua         newlibtest       patch          redoxfs          sdl2          terminfo
audiod           cpal          findutils       git            jansson       mdp         nulld            pcre           redox-ssh        sdl_gfx       termplay
autoconf         curl          freeciv         glib           kernel        mesa        openjazz         periodictable  relibc           sdl_image     timidity
automake         dash          freedoom        glium          keyboard-sfx  mesa_glu    openssl          perl           ripgrep          sdl_mixer     ttf-hack
bash             diffutils     freeglut        glutin         lci           miniserve   openttd          pixelcannon    rodioplay        sdl-player    userutils
binutils         dosbox        freetype        gnu-binutils   libc-bench    nasm        openttd-opengfx  pixman         rs-nes           sdl_ttf       uutils
bootloader       drivers       friar           gnu-grep       libffi        ncdu        openttd-openmsx  pkgutils       rust             sed           vim
ca-certificates  drivers-041   game-2048       gnu-make       libiconv      ncurses     openttd-opensfx  powerline      rust64           servo         vttest
cairo            duktape       games           gstreamer      libjpeg       ncursesw    orbdata          prboom         rust-cairo       shellstorm    webrender
cairodemo        eduke32       gawk            harfbuzz       liborbital    netdb       orbital          ptyd           rust-cairo-demo  smith         winit
cargo            exampled      gcc             hematite       libpng        netstack    orbterm          python         rustual-boy      sodium        xz
cleye            expat         gears           init           libsodium     netsurf     orbutils         qemu           schismtracker    sopwith       zerod
cmatrix          extrautils    generaluser-gs  installer      llvm          netutils    osdemo           randd          scummvm          ssh           zlib
contain          fd            gettext         ion            logd          newlib      pastel           readline       sdl              syobonaction
```

- Drivers

```
[~/work/repos/redox-workspace/redox.new/cookbook/recipes/drivers/source] [source::master] [last: 0s]
robin@vulcan $ ls
ahcid  bgad        Cargo.toml  filesystem.toml  initfs.toml  nvmed  ps2d      vboxd  xhcid
alxd   Cargo.lock  e1000d      ihdad            LICENSE      pcid   rtl8168d  vesad
```

The Redox Stack



- Redox subscribes to Plan 9's "everything is a file" philosophy but with a twist: In Redox everything is a URL

- This has resulted in a consistent, clean and flexible interface
  - No confusing semantic recursions: *"The rootfs is on a disk which contains device nodes at /dev including node sda which represents the disk containing the rootfs which…"*
  - No special file odditties: *"What's the size of /dev/null ?"*

- As opposed to traditional filesystem hierarchies, resources are distinguished by protocol based ***Schemes*** identified by URL
  - Eg: EHCI capable USB devices are accessed via the “usb:/ehci” scheme
  - Eg: Real files are accessed using the “file:/” scheme
- Each Scheme handles a section of the filesystem namespace
- Each Scheme is implemented in user-space with support from the kernel
- Applications communicate using URLs with each other, the system, with daemons and so on

- Written in Rust
- Provides user-space with primitives for
  - Physical memory access
  - Interrupt handling
  - Synchronisation with futexes
- Supports containerisation through scheme namespaces
  - Processes can be put into a “null” namespace
  - Doing so enables a per-process capability mode
  - Fine grained per-process access control
- SMP support
  - Simple “spread-out” scheduling at present

```
[~/work/repos/redox-workspace/redox.new/kernel/src] [kernel::aarch64] [last: 0s]
robin@vulcan $ loc
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                   150        23329         3690         1412        18227
 Assembly                 5          558           86            7          465
 C/C++ Header             1           49           11            6           32
--------------------------------------------------------------------------------
 Total                  156        23936         3787         1425        18724
--------------------------------------------------------------------------------
[~/work/repos/redox-workspace/redox.new/kernel/src] [kernel::aarch64] [last: 0s]
robin@vulcan $ loc ---exclude "aarch64"
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                   119        19617         3074         1097        15446
--------------------------------------------------------------------------------
 Total                  119        19617         3074         1097        15446
--------------------------------------------------------------------------------
[~/work/repos/redox-workspace/redox.new/kernel/src] [kernel::aarch64] [last: 0s]
robin@vulcan $ loc ---exclude "aarch64|acpi"
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                    96        13202         2021          991        10190
--------------------------------------------------------------------------------
 Total                   96        13202         2021          991        10190
--------------------------------------------------------------------------------
[~/work/repos/redox-workspace/redox.new/kernel/src] [kernel::aarch64] [last: 0s]
robin@vulcan $ loc ---exclude "aarch64|acpi|driver|test"
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                    94        13026         1991          981        10054
--------------------------------------------------------------------------------
 Total                   94        13026         1991          981        10054
--------------------------------------------------------------------------------
[~/work/repos/redox-workspace/redox.new/kernel/src] [kernel::aarch64] [last: 0s]
robin@vulcan $ loc ---exclude "aarch64|acpi|driver|test|syscall"
--------------------------------------------------------------------------------
 Language             Files        Lines        Blank      Comment         Code
--------------------------------------------------------------------------------
 Rust                    84         9947         1588          778         7581
--------------------------------------------------------------------------------
 Total                   84         9947         1588          778         7581
--------------------------------------------------------------------------------
```

- The question of virtualization and Redox
  - There is no support for virtualization at present
  - Current thinking
    - Support rebuilding software against relibc to run on Redox
    - Rather than support running unmodified software as is traditionally done

- Aiming to be a POSIX compliant C library written in Rust
  - Uses cbindgen for FFI’ing with C code
- Targets Redox and Linux environments
  - Enables running Linux apps under Redox
  - Enables running Redox apps under Linux
  - The latter uses an extension called Rine
- Relibc aims to be Linux compatible
  - At the syscall API level
  - At the syscall ABI level (for a given architecture)
- Rust linkage
  - The Rust compiler is built for the x86_64-unknown-redox triplet
  - Associated with relibc to support building Redox applications

The Arm porting saga

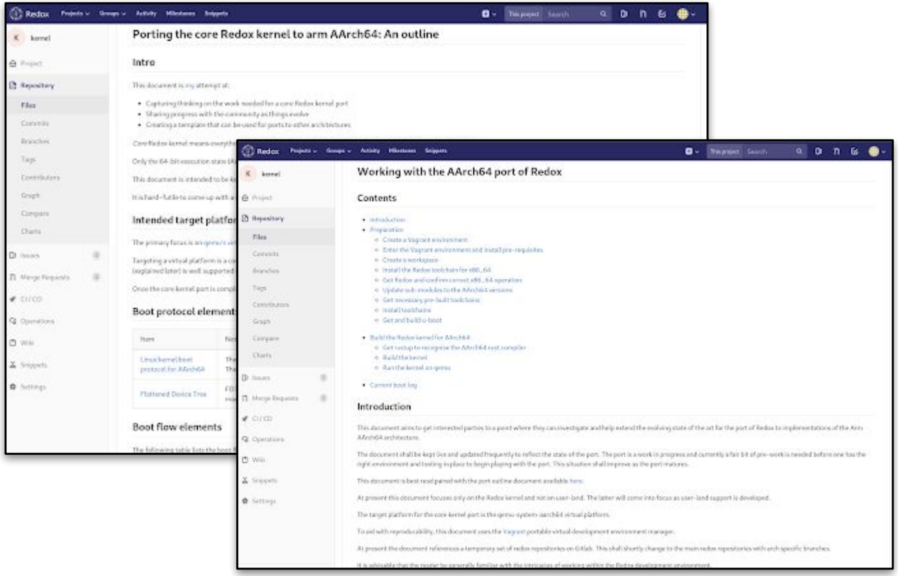
- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Studied the Redox x86_64 kernel port and asked a lot (a LOT) of questions on the redox kernel Mattermost channel
- Identified spots where x86_64 assumptions existed
- Decided to restrict the port to Armv8.0 and support only the AArch64 execution state
- Settled on qemu’s virt machine emulation for AArch64 as the initial platform target
  - Cortex-A57 x 1
  - 1 GB RAM
  - Generic timers
  - GICv2
  - PL011 UART
  - SP804 timers
  - PL031 RTC
  - E1000 ethernet
  - PCI-ECAM host controller

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Redox

- Wrote down the scope and published it on the Redox gitlab
- Began speaking with Arm legal eagles to get approvals



Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Studied the rust compiler toolchain at a high level (rustc, MIR, LLVM)
- Built it from source and played around with generating Linux app binaries and bare-metal code for AArch64
- Looked at the x86_64-unknown-redox support code in LLVM and wrote analogous bits to add support for the aarch64-unknown-redox triple
- Rinse-repeat until I rustup told me that it recognised this triple
- Lots of intermediate testing to verify that the generated code was sane
- Added support for the aarch64-unknown-redox triple to binutils and GCC

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Ran into trouble with Rust's #[thread_local] TLS decorator

```
#[thread_local]
static CPU_ID: AtomicUsize = ATOMIC_USIZE_INIT;
```

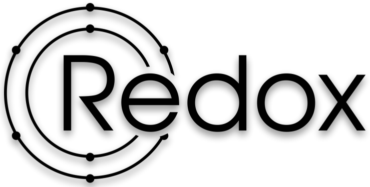
- Produced:

```
20:   d53bd041        mrs     x1, tpidr_el0
24:   8b000020        add     x0, x1, x0
```

- This is fine for user-mode TLS accesses at EL0 but the Redox kernel uses TLS for per-cpu data. Using tpidr_el0 at EL1 == boom

- I could have changed the kernel but was intrigued enough to try and fix LLVM (!)

- Modded LLVM to conditionally emit tpidr_el1 for any code compiled by the rust front-end using the "kernel" code-model. Problem solved!



- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Desired qemu's GDB stub to work with a multi-arch GDB client for both user-space and kernel space debugging
- Ran into trouble with GDB and EL1 access any attempt to "see" code at high virtual addresses would result in odd values
  - Seemingly impacted my bare-metal boot stub and even Linux (!)
  - Traced GDB
  - Traced GDB debug protocol
  - Banged my head on walls
  - Produced a reliable reproducer test case
  - Reported to GDB upstream
  - Worked with Linaro developers to resolve
- Came up with a kernel and user-land instruction tracing flow with qemu *(super useful!)*

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Stitched together a bootflow using the u-boot bootloader
- u-boot grew support for qemu's aarch64 virt machine
- My boot flow used qemu's tftp emulation and u-boot's ethernet capability to fetch a stub Redox kernel image from the host filesystem to the guest memory
- Got necessary environment info from u-boot through to the Redox kernel using standard Device Tree nodes ("/chosen")
- Verified GDB operation at the u-boot stage and the Redox kernel stage

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Replicated x86_64 kernel code structure (with a set of necessary mods for aarch64)
- Stubbed everything out
- Specified a linker script and got a linkable kernel image
- Verified that execution ends up in the kernel
- Started writing early init boot code in aarch64 assembly
  - Correct exception level transitioning
  - Virtual address range specification
  - Identity mapping the kernel code, data, stack, FDT images etc
  - Enabling the MMU using
    - 4 level page tables
    - 48-bit VAs
    - 2 MB Blocks
    - recursive paging
  - Created a Rust environment
  - Jumped to Rust code
- Verified everything with GDB

Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Kernel paging support:

- Fleshed out a recursive paging implementation for aarch64
  - Recursive paging gets you easy and performant page table manipulation
  - But wastes virtual address space
  - Not a concern at present
- Wrote code to map, unmap virtual address ranges
- Elf interpretation and section specific memory attribute mapping etc
- Got the kernel to successfully tear down the MMU mappings set up by the boot asm code and replace it with comprehensive paging with 4 KB pages
  - Mapped in the kernel code, data, stack, FDT image
  - Mapped in a diagnostic UART
- Redox kernel said “Hello World”!!!

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Basic driver set:

- Added basic drivers
  - Generic Interrupt Controller
  - Generic Timer
  - PL011 UART
  - PL031 RTC
  - SP804 Timer
- Verified operation with GDB and simple test code

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Added stack frame unwinding support
- Needed this to make sense of panic traces
- No symbol resolution support but was super useful even so

Scoping the port

Publishing the scope

Preparing a toolchain

Creating a debug flow

Creating a bootflow

Basic kernel bootstrap

Kernel paging support

Basic driver set

Stack frame unwinding

Relibc port

/bin/init bring-up

initfs bring-up

Context switching

Time keeping

FDT support

Live disk support

Login shell

Apps!

- Added aarch64 support to relibc
  - Syscall asm stubs
  - Syscall stack frame descriptions etc
- Lots of time spent trying to get this working properly with the rust toolchain
  - Redox community were super useful as always
- Mixed the relibc code into the main Redox kernel
- Wrote kernel side asm code to process syscalls
  - Syscall vectors
  - Context save and restore
  - Plugging into core kernel syscall machinery
- Got init to build and link successfully
- The stage was set to get user-land up!

Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Extended the x86_64 live disk to aarch64
  - Used it to build initfs + kernel image + live disk image blob
- Got the live disk image to load reliably with GDB's help
- Then tried to get init to be loaded into RAM and executed
- Gnashed and wailed for a long time before this finally worked
  - Lots of subtleties with ELF loading needed special care
  - Mapping Redox's higher level ELF section attributes to aarch64 page descriptor attributes was trickier than I had anticipated
  - Didn't have enough mutually exclusive spare bits between page tables and page descriptors
    - Needed to keep track of page and page table usage
  - Came up with an arcane hack
    - It worked!!!
- /sbin/init ran and said Hello!

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

initfs bring-up:

- Fleshed out essential syscall support code
  - fork, clone, dup, dup2 etc
  - Trickier than I imagined!
- Got initscript going
- Attempted to launch user-mode device drivers
  - Failed miserably
  - Found missing gaps in page table manipulation - filled
- Got to a point where a bunch of user-space contexts could be launched but had no context switching support yet

Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Context switching:

- Implemented context switching code
- Rinse repeat
  - User to kernel, user to user
  - Further syscall pathway enhancements
- Got multiple Contexts switching co-operatively
- Next step was asynchronous context switching

Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Time keeping:

- Added interrupt context save-restore support
- Hooked in the GIC
- Set up the Generic Timer to interrupt at 10ms intervals
- Added cheduler hooks for optional context switching
- Verified pre-emptive context switching across multiple contexts with simple tests

Scoping the port

Publishing the scope

Preparing a toolchain

Creating a debug flow

Creating a bootflow

Basic kernel bootstrap

Kernel paging support

Basic driver set

Stack frame unwinding

Relibc port

/bin/init bring-up

initfs bring-up

Context switching

Time keeping

FDT support

Live disk support

Login shell

Apps!

- FDT support for drivers
- Got timely help from the Redox community
  - They gave me a DT interpreter crate that could work without relying on the Rust standard library
- Used it to incrementally remove static assumptions from the drivers and replace them with information from the device tree (address maps, interrupt mappings etc)
- This is still ongoing

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

- Simplified the live disk support
  - Using qemu’s raw memory device emulation made it possible to pre-load RAM with the live disk image
  - Super fast booting! Great for rapid debug cycles.
  - Live disk image was weighing in at 256 MB - lots more work needed there but the raw memory device emulation made it a snap

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Login shell:

- Incrementally got getty going
- Got the Ion shell going
- Got to a prompt! :)
- Spent time refactoring
- Broke everything
- Spent time fixing and cleaning

Redox

- Scoping the port
- Publishing the scope
- Preparing a toolchain
- Creating a debug flow
- Creating a bootflow
- Basic kernel bootstrap
- Kernel paging support
- Basic driver set
- Stack frame unwinding
- Relibc port
- /bin/init bring-up
- initfs bring-up
- Context switching
- Time keeping
- FDT support
- Live disk support
- Login shell
- Apps!

Apps!:

- Got simpler tools like findutils etc working
- Added basic support for CPU identification and feature reporting
- Drank Beer. Lots of Beer.

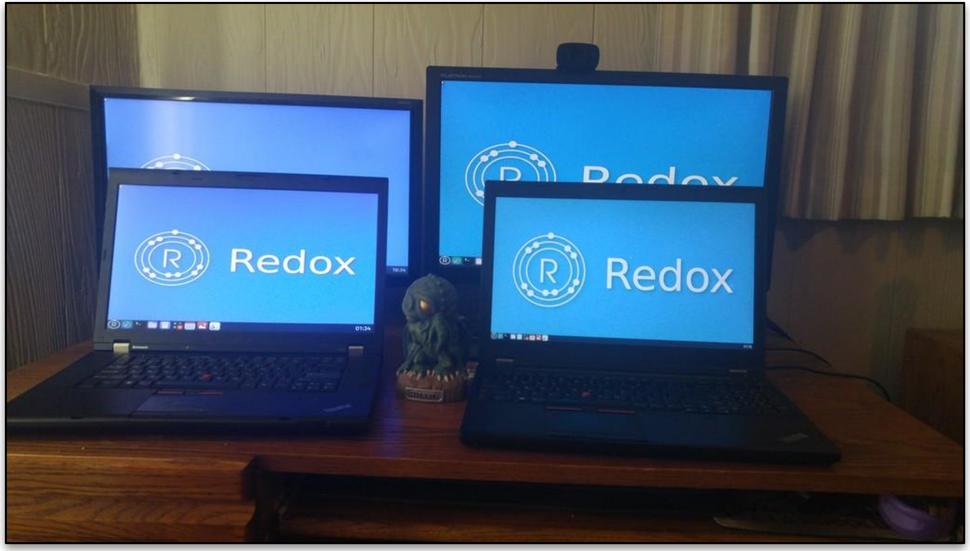
- Current status of the Arm port
  - Clean room exercise underway (read as “I’ve broken it at present”)
  - Code continually checked into “aarch64” branches for each Redox component on gitlab
  - Documentation revamp underway
  - Silicon bring-up underway on Raspberry Pi3 and Hikey970
    - Slower than expected but hope to resolve this soon ish

- General Redox roadmap items for 2019
  - Benchmarking infrastructure as a CI/CD gitlab target
  - Better SMP support
  - Priority based pre-emptive scheduler with pluggable policies
  - Move to lldb (external and self-hosted)
  - Bridge to Fuchsia and FreeBSD drivers
  - More native drivers
  - Dynamic loading + linking
  - IOMMU support
  - Device driver sandboxing with IOMMUs on Intel
  - OrbTk GUI toolkit refresh
  - Reincarnation server inspired by MINIX
  - RSoC 2019
  - Sweep contemporary designs for cool features to emulate

- Redox Arm roadmap items for 2019
    - Shadow the x86_64 port and achieve feature parity
        - Add SMP support
        - Add dynamic loading + linking support
        - Framebuffer support
        - Port the EFI OS loader to AArch64
    - Improve FDT support and convert more drivers
    - Complete WiP silicon bring-up (Raspberry Pi 3, Hikey970)
    - Switch from recursive to linear paging
    - GICv3, SMMU
    - Device driver sandboxing using SMMU

- The Redox community
  - Development is done on GitLab
  - Real-time discussion is done on Mattermost Chat
  - Other discussion is done on the Redox Forum on Discourse
  - Redox follows the Rust Code of Conduct
  - Redox has a Contributing Guide
  - All of this information can be found at https://redox-os.org

- Demo Time + Question Time



Redox