Redox OS + Arm

A Rust microkernel meets a RISC architecture

Jeremy Soller  
(System76)  
www.system76.com

Robin Randhawa  
(Arm)  
www.arm.com
Background

Arm System Software Architecture Team

**Safety Track**

Charter

“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

- Safety themed Arm hardware architecture
- Microkernel based system software composition
- Formally verified microkernels
- AUTOSAR
- Safety themed languages, run-times and tooling
Background

Arm System Software Architecture Team

Safety Track

Charter
“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

Arm
Open Source
Safety

- Safety themed Arm hardware architecture
- Microkernel based system software composition
- Formally verified microkernels
- AUTOSAR
- Safety themed languages, run-times and tooling

Hmm, Microkernels and the Arm architecture?
Background

Arm System Software Architecture Team

Safety Track

- Safety themed Arm hardware architecture
- Microkernel based system software composition
- Formally verified microkernels
- AUTOSAR
- Safety themed languages, run-times and tooling

Charter
“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

Hmm, Rust and the Arm architecture?
Background

Arm System Software Architecture Team

Safety Track

Charter
“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

- Safety themed Arm hardware architecture
- Microkernel based system software composition
- Formally verified microkernels
- AUTOSAR
- Safety themed languages, run-times and tooling

Hmm, Rust + Microkernels and the Arm architecture?
Background

Arm System Software Architecture Team

Safety Track

Charter
“Promote the uptake of Arm IP in safety critical domains using open source software as a medium”

Let’s write a microkernel in Rust!

- Safety themed Arm hardware architecture
- Microkernel based system software composition
- Formally verified microkernels
- AUTOSAR
- Safety themed languages, run-times and tooling
Objectives

● Test the claim that Rust is a capable systems programming language
  ○ Bootloaders, kernels, drivers etc
● Create a sandbox that has a microkernel implemented in Rust
  ○ For enabling Arm architecture explorations

And so I began writing a microkernel....
And then I found Redox OS!

https://www.redox-os.org
What is Rust?

“Rust is an open source systems programming language that:

- runs blazingly fast
- prevents segfaults and
- guarantees thread safety”

https://www.rust-lang.org

```rust
def main() {
    println!("Hello World!");
}
```
Mutability

Data in Rust is **immutable** by default!

Data has to be explicitly qualified as mutable using the `mut` type qualifier

This significantly reduces pitfalls associated with shared data manipulation

```rust
let var: u32 = 0;
var += 1;

error[E0384]: cannot assign twice to immutable variable `var`
  --> foo.rs:3:5
   | 2 | let var: u32 = 0;
   |    --- first assignment to `var`
   3 | var += 1;
   |     ^^^^^^^ cannot assign twice to immutable variable

let mut var: u32 = 0;
var += 1;
```
Ownership

Mutable types can only be used by one scope

Immutable types can be used in multiple scopes

No types can be used after they exit the scope, unless they are moved to another scope
Ownership of the type moves from the calling scope to the callee scope
References allow sharing of the type but the same mutability rules apply

These rules are checked by the compiler

Therefore none of the following occur. Also applicable in concurrent running threads:

- Use before initialization or after free errors
- Null dereference errors
- Double free errors
- Ignoring error values
- Stack overflows
- Data races
- Memory leaks
Type width clarity

All type names encode the required type size

There is no ambiguity around the size of ints, longs etc across multiple processor architectures

<table>
<thead>
<tr>
<th>Signed ints:</th>
<th>i8  i16 i32 i64 i128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned ints:</td>
<td>u8  u16 u32 u64 u128</td>
</tr>
<tr>
<td>Floats:</td>
<td>f32  f64</td>
</tr>
</tbody>
</table>
Threads are a 1st class language item

use std::thread;

static NTHREADS: i32 = 10;

// This is the `main` thread
fn main() {
    // Make a vector to hold the children which are spawned.
    let mut children = vec![];

    for i in 0..NTHREADS {
        // Spin up another thread
        children.push(thread::spawn(move || {
            println!("this is thread number ", i);
        }));
    }

    for child in children {
        // Wait for the thread to finish. Returns a result.
        let _ = child.join();
    }
}
Rust has a rich standard library

**Containers and Collections:** Maps, sets, linked lists, dictionaries and other typical collection types

**Platform abstractions and I/O:** Files, TCP, UDP, threads, shared-memory types

**Goodies from functional languages:** Closures, Iterators, Generators, Pattern matching
Synch primitives are *always* coupled with the data they marshal accesses to.

To access the data you *must* take the lock because the lock *owns* the data.

Not possible to forget to take a lock!

This is checked at compile time.
Rust enables localising of unsafe code

‘Unsafe’ code blocks tell the compiler to not perform checking of the code within
These are typically used for low level access or when unsafe type casts are absolutely needed
All the unsafe is clearly labelled and easily available for static analysis

```rust
unsafe {
    asm!("wfi" ::::);
}
```
Atomic Types

Rust has Atomic types

Atomic types provide primitive shared-memory communication between threads, and are the building blocks of other concurrent types

Load/Stores to Rust Atomic types permit desired memory consistency specification

(Relaxed, Release, Acquire, AcqRel, SeqCst)

```rust
use std::sync::atomic::{AtomicUsize, Ordering};
let spinlock = Arc::new(AtomicUsize::new(1));

let spinlock_clone = spinlock.clone();
let thread = thread::spawn(
    move || {
        spinlock_clone.store(0, Ordering::SeqCst);
    });

// Wait for the other thread to release the lock
while spinlock.load(Ordering::SeqCst) != 0 {}
Concurrent Rust programs tend to be faster and safer than their C and C++ counterparts

Compile time ownership checks mean that a lot of the traditional synchronisation boiler-plate in C and C++ is not needed at run-time

Less boiler-plate == faster code

As an simple example, checkout ripgrep a rust implementation of grep
(It’s faster than {grep, ag, git grep, ucg, pt, sift}!!!)
Redox OS

- Unix
- GNU
- Linux
- Honorable Mentions
- Redox
- Redox Roadmap
- Redox Community
- Redox Microkernel
- Redox Userspace
- Example Driver
Unix

- Name derived from Multics - an earlier OS
- Development began in 1969
- Modular philosophy - "Do One Thing and Do It Well"
- Multi-user and multi-tasking
- Simple filesystem with a unified namespace
- Simple system calls - "Everything is a file"
- Powerful shell where most programs can be scripted
- Portable - written in C for many platforms
- Divine - time is seconds since the holy birth of Unix
- Creator is 75 years old, with a beard to match

Ken Thompson
GNÜ

- GNU's not Unix
- Development began in 1984
- Free Software - Free as in Freedom
- Developed critical free software licenses
- Started with free rewrites of many Unix components
- Most projects are written in C
- Developed a microkernel, GNU Hurd
- GNU maintains a large number of software projects
- Usually paired with Linux, GNU+Linux
- Creator is 65 years old, with a moderate beard
Linux

- Linus + Unix = Linux
- Development began in 1991
- Free Unix-like kernel implementation
- Written in C
- Is the de facto kernel for GNU
- Has dominated most computing markets
- Has out-of-the-box support for most devices
- Creator is 48 years old, with some stubble

Linus Torvalds
Honorable Mentions

- FreeBSD
  - Capsicum - capability-based security
- Minix 3
  - Reincarnation Server - restart failed drivers
  - Grants - kernel-facilitated memory-mapped IPC
- NetBSD
  - pkgsrc - package management system
  - rumpkernel - portable, userspace drivers
- Plan 9
  - Everything is actually a file
  - Simple set of system calls
- SmartOS
  - Zones - OS-level virtualization
Unix Timeline (Not Simplified or Legible)
Redox

- Redox sounds like Unix and Rust is a Redox reaction
- Development began in 2015
- Only contains free software - MIT license
- Microkernel - all drivers are in userspace
- Says "Everything is a file", and really means it
- Keeps Unix system calls, but reduces the set significantly
- Uses Rust to improve reliability and reduce vulnerabilities
- Rust rewrite of most Unix components
- Provides complete C library for compatibility
- Creator is 26 years old, and is in front of you
Redox Roadmap

Overall Ambition
- Desktop/Workstation as initial target
- Embedded systems as the current focus
- Servers for the future

Specific Items
- Improvements for current drivers
- Implementing more drivers
- Opt-in watchdog for programs
- Better SMP scheduling
Redox Community

- Development is done on our [GitLab](https://gitlab.com)
- Real-time discussion is done on our [Chat](https://discord.gg/redox-os)
- Other discussion is done on our [Forum](https://forum.redox-os.org)
- We follow the [Rust Code of Conduct](https://rust-lang.org/faq.html#code-of-conduct)
- We have a [Contributing Guide](https://redox-os.org/contributing)
- All of this information can be found at [https://redox-os.org](https://redox-os.org)
Redox Microkernel

- Written in Rust to provide some compile-time verification
- Introduces a concept of kernel or userspace implemented filesystems called schemes
- Each scheme handles a section of the filesystem namespaces
- Limits system calls to those required for managing processes and communicating with schemes
- Uses memory mapping to increase scheme performance and reduce context switching
- Provides primitives for interrupt handling and physical memory access for drivers
- Supports containerization through scheme namespaces
- Processes may enter a capability mode by entering the null namespace
- 20,000 lines of code (5,000 of which is optional for ACPI, and might move to userspace)
- May already be feature complete, if all remaining features can be implemented in userspace
Redox Userspace

- Disk drivers
- Filesystems
- Network drivers
- Network stack
- Graphics drivers
- Graphics stack
- Everything else
Example Driver

Drivers in Redox are userspace applications. They may exist in the initfs, prior to filesystem availability, as disk drivers do. Most drivers are loaded after the disk driver is loaded and the filesystem is available.

A driver in Redox typically utilizes an interrupt vector, and handles a scheme. These are done using event-based I/O, such that one thread can manage multiple interrupt vectors, schemes, and other related files.

Most of this initialization can take place in the main function before finally entering into the "null namespace", whereby the driver cannot open any more files.

We will now look at a driver that services an IRQ and handles a scheme.

https://gitlab.redox-os.org/redox-os/exampled
Redox Runs on Real Hardware
Redox OS for Arm

● Targets the AArch64 execution state
● 3 WIP platform targets
  ○ qemu-system-aarch64 - virt machine platform
  ○ Hikey620 board
  ○ Hikey960 board
Porting journey

- Published a plan
- Ported the toolchain
- Setup a boot flow
- Ported the kernel
- Ported drivers
- Today
Porting journey

- Published a plan
- Ported the toolchain
- Setup a boot flow
- Ported the kernel
- Ported drivers
- Today

---

**Intended target platform**

The primary focus is on a toolchain/platform combination for the Arm64 architecture.

**Boot protocol elements**

- Linux kernel boot protocol
- Flatfile Device Tree

**Boot flow elements**

- The following is added to the boot flow:

---

**Working with the Arm64 port of Redox**

**Contents**

- Introduction
- Boot protocol
- bootloader
- Flatfile Device Tree
- Linux kernel boot protocol

---

**Introduction**

This document aims to get interested parties to a point where they can investigate and help extend the existing work for the Arm64 port of the Redox kernel.
Porting journey

Published a plan

Ported the toolchain

Setup a boot flow

Ported the kernel

Ported drivers

Today

**rustc + LLVM**
- Added ‘aarch64-unknown-redox’ triplet support
- Added support for thread_local (TLS) at EL1 (tpidr_el1) for the AArch64 llvm kernel code model (--code-model=kernel)

**binutils**
- Added ‘aarch64-unknown-redox’ triplet support

**newlib**
- Redox syscall veneers

**GDB**
- Found a bug with high Virtual Address accesses (done via TTBR1). Reported upstream. Now fixed.
Porting journey

Published a plan ➔ Ported the toolchain ➔ Setup a debug flow ➔ Ported the kernel ➔ Ported drivers ➔ Today

**u-boot**
- $ make qemu_arm64_defconfig && make

**qemu**
- $ qemu-system-aarch64 -serial mon:stdio -smp 4 -machine virt -cpu cortex-a57 -m 1024 -netdev user,id=net0,tftp=build -net nic,model=e1000,netdev=net0 -nographic -vga none -drive if=none,file=build/filesystem.bin,id=mydisk -bios u-boot.bin -S -s
- $ aarch64-elf-gdb
  - telnet localhost:1234
  - b *__redox_early_init
  - c
Porting journey

Published a plan → Ported the toolchain → Setup a boot flow → Ported the kernel → Ported drivers → Today

Early init code
- Optional EL2 to EL1 handoff
- Early page table setup for diagnostic UART access and for kernel entry
- DTB and environment relay to the kernel

Arch specific kernel code
- MMU setup:
  - Page tables, page descriptors, access control, recursive paging
- Context management:
  - Context creation (ELF parsing-loading, Fork, Clone, Exec)
  - Context switching
  - Signal dispatch
- User-space support:
  - Crafted /bin/init context
- Syscalls
  - Exception entry, decode, despatch
Porting journey

- In kernel driver “stubs”
  - Generic timers
  - PL01x UARTs
  - GICv2

- User-space drivers
  - PCI (A simple accessor for qemu’s ECAM model)
  - AHCI (For SATA disk access to a RedoxFS formatted filesystem)
Porting journey

Published a plan → Ported the toolchain → Setup a boot flow → Ported the kernel → Ported drivers → Today

- Currently bringing up multi-user login and shell
- Coming up
  - Device Tree scheme
  - Convert drivers to use this scheme
  - Frame buffer driver for GUI bringup
  - Hikey620 and Hikey960 bringup
  - Documentation update on gitlab
  - SMP support
  - Parity with the x86_64 port
The End