OWNERSHIP IS THEFT: EXPERIENCES BUILDING AN EMBEDDED OS IN RUST

Amit Levy   Michael P Andersen   Bradford Campbell   David Culler
Prabal Dutta   Branden Ghena   Philip Levis   Pat Pannuto

October 4th, 2015

OWNERSHIP IS THEFT: EXPERIENCES BUILDING AN EMBEDDED OS IN RUST

Amit Levy Michael P Andersen Bradford Campbell **David Culler**
Prabal Dutta Branden Ghena **Philip Levis** Pat Pannuto

October 4th, 2015

- TinyOS about 15 years old

- TinyOS about 15 years old
- Lots of changes in embedded applications hardware since

- TinyOS about 15 years old
- Lots of changes in embedded applications hardware since
- TinyOS written in nesC (a dialect of C)

- TinyOS about 15 years old
- Lots of changes in embedded applications hardware since
- TinyOS written in nesC (a dialect of C)
- Decided to revisit the design of an embedded OS and write it in a safe language.

"When we ported TinyOS from C to nesC it only took a few weeks!"

But here we are a year later...

But here we are a year later…

We didn't take into account how much our choice of language would affect our system's design.

It can be convenient to think of operating system design and language design as independent.

It can be convenient to think of operating system design and language design as independent.

OS Designer

"If I simply port my bug-free C code into a safe language, my OS will be safe!"

It can be convenient to think of operating system design and language design as independent.

OS Designer

"If I simply port my bug-free C code into a safe language, my OS will be safe!"

PL Designer

"My new language abstraction provides safety, and all you have to do is use it ubiquitously in your system!"

It can be convenient to think of operating system design and language design as independent.

OS Designer

"If I simply port my bug-free C code into a safe language, my OS will be safe!"

PL Designer

"My new language abstraction provides safety, and all you have to do is use it ubiquitously in your system!"

But we can't forgot that system design and language design are actually ***co-dependent***!

- Background
    - Microcontrollers
    - Tock: an new embedded OS
    - Rust
- Two Challenges
- Conclusion

BACKGROUND - MICROCONTROLLERS

Who uses Microncontrollers?

Not Your Grandchildren's Processor

- Very little memory
  - Range 16-512KB RAM
- Crashes are particularly expensive:
  - Cannot assume user intervation (no screen or keyboard)
  - High stakes: implanted medical devices, home automation...
- Limited hardwared protection:
  - Specifically, no virtual memory

Limited Hardware Motivates Language Choice

- Too little memory to use processes for isolation
- Crashes are expensive, so we should catch as many bugs as possible at compile-time.

BACKGROUND – TOCK

Tock is an embedded operating system we’ve been building for about a year.

Why a *New* OS?

- Existing systems: TinyOS, Contiki, FreeRTOS
- New requirements
- New hardware
- New programming language(s)

New Requirements

- Traditional embedded operating systems were design for single app devices.
  - Software updated never or rarely.
  - Everything is trusted, including the app.
- New applications are *platforms*.
  - Software updates
  - Third-party apps
- Kernel extensions should really be isolated
  - Drivers may come from a variety of sources.
  - …like Linux drivers but for your defibrillator!

Microcontrollers have improved drastically

- From 16-bit arch @ 6Mhz to 32-bit arch @ 48Mhz
- More busses, more timers, AES encryption, etc in hardware
- A system-call interfance no longer a performance barrier

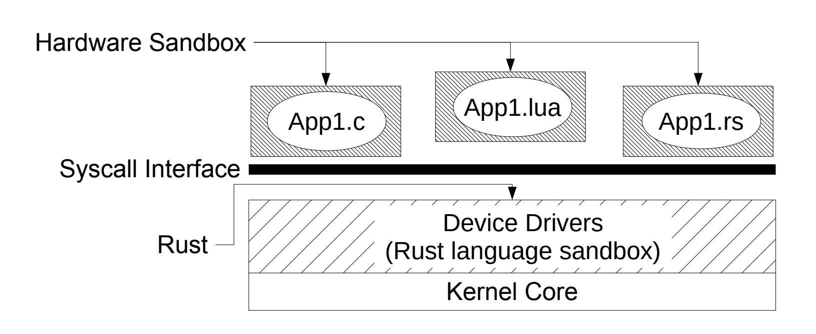
Some limited support for hardware protection

- Memory Protection Unit in ARM Cortex-M series

Rust, a new *safe* language without the runtime overhead

- Memory and type safety
- Eliminate large classes of bugs at *compile time*
- Strong type-system can allow component isolation
- Low-level primitives can enable rich security systems

Tock Architecture



BACKGROUND - RUST

Why Rust?

Two distinguishing properties from other safe languages:

- Enforces memory and type safety without a garbage collector
- Explicit separation of trusted vs. untrusted code
  - Untrusted code is strictly bound by the type system
  - Trusted code can circumvent the type system

Rust avoids the runtime overhead of garbage collection by using *ownership* to determine when to free memory at *compile-time*.

Ownership for Safety

Each Value has a Single Owner

Key Property

When the owner goes out of scope, we can deallocate memory for the value.

Each Value has a Single Owner

Key Property

When the owner goes out of scope, we can deallocate memory for the value.

Memory for the value `43` is allocated and bound to the variable `x`.

```
{
  let x = 43
}
```

When the scope exits, `x` is no longer valid and the memory is "freed"

Each Value has a Single Owner

Single owner means *no aliasing*, so values are either copied or moved between variables.

Each Value has a Single Owner

Single owner means *no aliasing*, so values are either copied or moved between variables.

This is an error:

```
{
  let x = Foo::new();
  let y = x;
  println("{}", x);
}
```

because `Foo::new()` has been moved from `x` to `y`, so `x` is no longer valid.

Functions must explicitly hand ownership back to the caller:

```
fn bar(x: Foo) -> Foo {
  // Do stuff
  x // <- return x
}
```

Or can use **borrows**: a type of reference which does not invalidate the owner.

```
fn bar(x: &mut Foo) {
  // Do stuff
  // the borrow is implicitly released.
}

fn main() {
  let mut x = Foo::new();
  bar(&mut x);
  println!("{}", x); // x still valid
}
```

Borrows

Borrows are resolved at compile-time, with some constraints:

- A value can only be *mutably* borrowed if there are no other borrows of the value.
- Borrows cannot outlive the value they borrow.
- Values cannot be moved while they are borrowed.

CHALLENGES

1. Ownership vs. Cycles
2. Closures as Callbacks
3. Memory allocation for long-lived closures (see paper)

CHALLENGES - OWNERSHIP VS. CYCLES

Ownership vs. Cycles

Circular references between OS modules are ubiquitous.

`RadioDriver` has to notify `IPStack` of incoming packets, while `IPStack` uses the `RadioDriver` to send packets.

```
impl IPStack {
  fn send(&mut self, packet) {
    packet.concat_ip_header_to_pkt();
    self.radio.send(packet);
  }
}
impl RadioDriver {
  fn on_receive(&mut self, packet) {
    self.ip_stack.incoming(packet);
  }
}
```

Several Possible Solutions

1. Combine mutually dependent modules
2. Message passing instead of shared state
3. Use unsafe language features
4. Use explicitly aliasable reference types

Combine Mutually Dependent Modules

```
impl IPStackAndAlsoRadioDriver {
  fn send(&mut self, packet) {
    packet.concat_ip_header_to_pkt();
    // write packet to radio
  }

  fn on_receive(&mut self, packet) {
    // just handle here
  }
}
```

- No modularity/extensibility.
- Least upper bound of trust

Message Passing

```
impl IPStack
  fn send(&mut self, packet) {
    packet.concat_ip_header_to_pkt();
    self.packet_out_chan.send(packet);
  }

  fn do_work() {
    self.process_pkt(self.packet_in_chan.recv());
  }
}
```

Might work if you're willing to have threads and dynamically allocate or block.

Use `unsafe` Features

```
pub static mut IPSTACK : IPStack = ...;

impl UDP {
  fn send(&self, packet) {
    unsafe { // Unsafe to borrow a mutable static
        IPSTACK.send(packet);
    }
  }
}
```

Requires including trusting virtually every component in the system.

Explicitly Sharable Reference Types

- A small bit of `unsafe` to make a reference type that is aliasable.
    - E.g. `core::cell::RefCell` dynamically checks borrow rules
- Loose compile-time guanratees

Explicitly Sharable Reference Types

This is what we are using.

- Had to completely eliminate concurrency in the kernel
- Be very cautious about who gets shared references.
- Enqueue all interrupts to run in main kernel thread
  - Avoid any work in interrupt handlers
  - Lose hardware interrupt priorities
  - Probably OK for performance, although still need to validate

CHALLENGES - CLOSURES AS CALLBACKS

Problem

Asyncornous code in C usually uses one of two mechanism:

- State-machines
- Function pointers + stack ripping

Both are:

- Difficult to read/write/maintain
- Bug prone

Closures to the Rescue?

Event-driven application languages (e.g. JavaScript) address this problem by specifying callbacks as closures at the callsite.

```
var count = 0;

setInterval(function() {
  console.log(count + " clicks");
}, 2000);

onClick(function() {
  count += 1;
});
```

Closures to the Rescue?

Event-driven application languages (e.g. JavaScript) address this problem by specifying callbacks as closures at the callsite.

```
var count = 0;

setInterval(function() {
  console.log(count + ” clicks”);
}, 2000);

onClick(function() {
  count += 1;
});
```

Rust has closures… so we can just do that!

Ownership Strikes Again

In Rust, closures have two options:

- Take ownership of closed over variables
- Complete before returning to the caller

```
let mut x = 0;

setInterval(move ||
  println!("{} clicks", x);
}, 2000);

// x is no longer valid in this context, and we
// cannot create the closure for onClick
```

Result: Oversharing vs. Undersharing

We have to carefully partition state between caller and callbacks. But that's very hard to get right.

Overshare with the callback:

```
// No other code can access any LEDs
setTimeout(|| {
  leds.activityToggle();
}, 2000);
```

Partition resources into tiny interfaces that are hard to manage:

```
setTimeout(|| {
  activityLed.toggle();
}, 2000);
```

CONCLUSION

Conclusion

- Embedded systems have unique requirements
- Rust seems well-suited for a resource constrained OS
    - Type- and memory- safe with no garbage collector
- Tock: an embedded OS
    - Comines hardware and language protection
- Ownership is both the hero and the villan in our story
- Closures not as powerful as we're familiar from other languages

In the paper

- Proposal to expose threads in the type system

Conclusion

System design and language design are *not* independent.

Conclusion

System design and language design are *not* independent.

Open Questions

How should we design operating systems to best leverage safe languages?

Conclusion

System design and language design are *not* independent.

Open Questions

How should we design operating systems to best leverage safe languages?

How can we design safe languages to better target low-level systems?

Conclusion

System design and language design are *not* independent.

Open Questions

How should we design operating systems to best leverage safe languages?

How can we design safe languages to better target low-level systems?

Thanks!

`http://tockos.org`
`http://github.com/helena-project/tock`