Rust for Linux

Error handling with the ?-operator

-

The problem of ‘silent’ errors

Kangrejos 2024

Dirk Behme <dirk.behme@gmail.com>

Abstract:

Using the ?-operator is a common and convenient way for error handling in Rust. So it is used in Rust for Linux (RFL) a lot, as well. However, in the error case the result in Rust for Linux is different to ‘normal’ Rust. In ‘normal’ Rust the error case results in a nice error message which helps analyzing the error. While in RFL the error details are not output anywhere, usually it errors silently. This makes any analysis (a) if there is an error and (b) where the error is without further tools (e.g. debugger) nearly impossible. Present some examples for this and discuss possible improvements.

(skip this page at presentation)

Simple example

```
fn larger_three(val: u32) -> Result {

    if val > 3 {
        return Err(EINVAL);
    }

    return Ok(())
}
```

Usage

```
larger_three(1)?;
larger_three(2)?;
larger_three(3)?;
```

Simple example

```
fn larger_three(val: u32) -> Result {

    if val > 3 {
        return Err(EINVAL);
    }

    return Ok(())
}
```

Usage

```
larger_three(1)?;
larger_three(2)?;
larger_three(3)?;
```

OK

Simple example (unchanged)

```
fn larger_three(val: u32) -> Result {

    if val > 3 {
        return Err(EINVAL);
    }

    return Ok(())
}
```

Changed usage

```
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
```

Simple example (unchanged)

```
fn larger_three(val: u32) -> Result {

    if val > 3 {
        return Err(EINVAL);
    }

    return Ok(())
}
```

Changed usage

```
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
```

Runtime result ‘normal’ Rust:

```
Error: 22
```

Note: EINVAL == 22

Simple example (unchanged)

```
fn larger_three(val: u32) -> Result {

    if val > 3 {
        return Err(EINVAL);
    }

    return Ok(())
}
```

Changed usage

```
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
```

Runtime result ‘normal’ Rust:

```
Error: 22
```

Runtime result RFL:

Note: EINVAL == 22

Note: Example code put to rust_minimal.rs and executed compiled in at boot time. No error output on console

Conclusion

Just using the ?-operator in RFL like in

```
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
```

the error case doesn’t give any error message to the console. What

a) makes the user fail to get *any* idea that there is an error *at all*

b) makes any analysis impossible in which module in which line the error happened

Conclusion

Just using the ?-operator in RFL like in

```
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
```

the error case doesn’t give any error message to the console. What

a) makes the user fail to get *any* idea that there is an error *at all*

b) makes any analysis impossible in which module in which line the error happened

Do we have options for doing better?

Let’s try expect()

```
larger_three(1).expect("larger_three(1)");
larger_three(4).expect("larger_three(4)");
larger_three(3).expect("larger_three(3)");
```

Let’s try expect()

```
larger_three(1).expect("larger_three(1)");
larger_three(4).expect("larger_three(4)");
larger_three(3).expect("larger_three(3)");
```

Runtime result:

```
rust_kernel: panicked at samples/rust/rust_minimal.rs:39:21:
larger_three(4): EINVAL
...
```

Nice!

Let's try expect()

```
larger_three(1).expect("larger_three(1)");
larger_three(4).expect("larger_three(4)");
larger_three(3).expect("larger_three(3)");
```

Runtime result:

```
rust_kernel: panicked at samples/rust/rust_minimal.rs:39:21:
larger_three(4): EINVAL
...
```

Nice! But …

```
...
------------[ cut here ]------------
kernel BUG at rust/helpers.c:51!
Internal error: Oops - BUG: 00000000f2000800 [#1] PREEMPT SMP
Modules linked in:
CPU: 1 PID: 1 Comm: swapper/0 Not tainted 6.10.0-rc1-arm64
Hardware name: aarch64 board (DT)
...
---[ end trace 0000000000000000 ]---
Kernel panic - not syncing: Oops - BUG: Fatal exception
SMP: stopping secondary CPUs
Kernel Offset: disabled
CPU features: 0x04,00001041,20100000,0200401b
Memory Limit: 6016 MB
Rebooting in 3 seconds..
```

Conclusion

Using expect() in RFL like in

```
larger_three(1).expect("larger_three(1)");
larger_three(4).expect("larger_three(4)");
larger_three(3).expect("larger_three(3)");
```

the error case gives the failing module with file name, line number and error details/code.
This is what we are looking for!

But:

The system panics with all its consequences we don’t want!

Conclusion

Using expect() in RFL like in

```
larger_three(1).expect("larger_three(1)");
larger_three(4).expect("larger_three(4)");
larger_three(3).expect("larger_three(3)");
```

the error case gives the failing module with file name, line number and error details/code.
This is what we are looking for!

But:

The system panics with all its consequences we don’t want!

Do we have options for doing better?

Let’s try ScopeGuard

```
let error_log = ScopeGuard::new(|| pr_err!("Error: larger_three() returned with error\n"));
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
error_log.dismiss();
```

Let’s try ScopeGuard

```
let error_log = ScopeGuard::new(|| pr_err!("Error: larger_three() returned with error\n"));
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
error_log.dismiss();
```

Runtime result:

```
rust_minimal: Error: larger_three() returned with error...
```

Much better, we at least get an info about an error and about the module. And no system crash.

But which line??

Let’s try ScopeGuard

```
let error_log = ScopeGuard::new(|| pr_err!("Error: larger_three() returned with error\n"));
larger_three(1)?;
larger_three(4)?;
larger_three(3)?;
error_log.dismiss();
```

Runtime result:

```
rust_minimal: Error: larger_three() returned with error...
```

Much better, we at least get an info about an error and about the module. And no system crash.

But which line??

Do we have options for doing better?

Let’s try inspect_err() and helper macro

```
larger_three(1).inspect_err(|e| err_info!(e))?;
larger_three(4).inspect_err(|e| err_info!(e))?;
larger_three(3).inspect_err(|e| err_info!(e))?;

#[macro_export]
macro_rules! err_info {
    ($e:expr) => {
        pr_err!("Error in {}:{}: {:?}n", file!(), line!(), $e)
    };
}
```

Let’s try inspect_err() and helper macro

```
larger_three(1).inspect_err(|e| err_info!(e))?;
larger_three(4).inspect_err(|e| err_info!(e))?;
larger_three(3).inspect_err(|e| err_info!(e))?;

#[macro_export]
macro_rules! err_info {
    ($e:expr) => {
        pr_err!("Error in {}:{}: {:?}n", file!(), line!(), $e)
    };
}
```

Runtime result:

```
rust_minimal: Error in samples/rust/rust_minimal.rs:40: EINVAL
```

Let’s try inspect_err() and helper macro

```
larger_three(1).inspect_err(|e| err_info!(e))?;
larger_three(4).inspect_err(|e| err_info!(e))?;
larger_three(3).inspect_err(|e| err_info!(e))?;

#[macro_export]
macro_rules! err_info {
    ($e:expr) => {
        pr_err!("Error in {}:{}: {:?}n", file!(), line!(), $e)
    };
}
```

Runtime result:

```
rust_minimal: Error in samples/rust/rust_minimal.rs:40: EINVAL
```

This is what we are looking for!

Let’s try inspect_err() and helper macro

```
larger_three(1).inspect_err(|e| err_info!(e))?;
larger_three(4).inspect_err(|e| err_info!(e))?;
larger_three(3).inspect_err(|e| err_info!(e))?;

#[macro_export]
macro_rules! err_info {
    ($e:expr) => {
        pr_err!("Error in {}:{}: {:?}n", file!(), line!(), $e)
    };
}
```

Runtime result:

```
rust_minimal: Error in samples/rust/rust_minimal.rs:40: EINVAL
```

This is what we are looking for!

But not always?

Let’s have a look: What do we have in C?

1. **Not all errors are verbose - not all errors are silent**

Let’s have a look: What do we have in C?

1. **Not all errors are verbose - not all errors are silent**

   In C the error handling is done ‘manually’ (like with match) what allows a case-by-case decision if the error handler shall be verbose or not

```
ret = something_returning_success_or_error();
if (ret == error)
      return –ERRVAL;
```

   versus

```
ret = something_returning_success_or_error();
if (ret == error) {
      printk(KERN_ERR “%s: Error %d\n”, __func__, ret);
      return ret;
}
```

What do we have in C? contd.

2. **Message/error levels (debug, info, crit …) and dynamic/runtime configuration**

What do we have in C? contd.

2. **Message/error levels (debug, info, crit …) and dynamic/runtime configuration**

It’s available for RFL:

But not in combination with the ?-operator.
We are back with ‘manual’ match error handling, then?

From: Danilo Krummrich <dakr@redhat.com>
To: gregkh@linuxfoundation.org, rafael@kernel.org,
bhelgaas@google.com, ojeda@kernel.org, alex.gaynor@gmail.com,
wedsonaf@gmail.com, boqun.feng@gmail.com, gary@garyguo.net,
bjorn3_gh@protonmail.com, benno.lossin@proton.me,
a.hindborg@samsung.com, aliceryhl@google.com, airlied@gmail.com,
fujita.tomonori@gmail.com, lina@asahilina.net,
pstanner@redhat.com, ajanulgu@redhat.com, lyude@redhat.com,
robh@kernel.org, daniel.almeida@collabora.com
Cc: rust-for-linux@vger.kernel.org, linux-kernel@vger.kernel.org,
linux-pci@vger.kernel.org,
Wedson Almeida Filho <wedsonaf@google.com>,
Danilo Krummrich <dakr@redhat.com>
Subject: [PATCH v2 06/10] rust: add `dev_*` print macros.
Date: Wed, 19 Jun 2024 01:39:52 +0200 [thread overview]
Message-ID: <20240618234025.15036-7-dakr@redhat.com> (raw)
In-Reply-To: <20240618234025.15036-1-dakr@redhat.com>

From: Wedson Almeida Filho <wedsonaf@google.com>

Implement `dev_*` print macros for `device::Device`.

They behave like the macros with the same names in C, i.e., they print
messages to the kernel ring buffer with the given level, prefixing the
messages with corresponding device information.

Signed-off-by: Wedson Almeida Filho <wedsonaf@google.com>
Signed-off-by: Danilo Krummrich <dakr@redhat.com>
---
rust/kernel/device.rs | 319 ++-
rust/kernel/prelude.rs | 2 +
2 files changed, 320 insertions(+), 1 deletion(-)

```
diff --git a/rust/kernel/device.rs b/rust/kernel/device.rs
index e445e87fb7d7..058767339a64 100644
--- a/rust/kernel/device.rs
+++ b/rust/kernel/device.rs
@@ -8,7 +8,10 @@
     bindings,
     types::{ARef, Opaque},
 };
-use core::ptr;
+use core::{fmt, ptr};
+
+#[cfg(CONFIG_PRINTK)]
+use crate::c_str;

 /// A reference-counted device.
 ///
@@ -79,6 +82,110 @@ pub unsafe fn as_ref<'a>(ptr: *mut bindings::device) -> &'a Self {
         // SAFETY: Guaranteed by the safety requirements of the function.
         unsafe { &*ptr.cast() }
     }
+
+    /// Prints an emergency-level message (level 0) prefixed with device information.
+    ///
+    /// More details are available from [`dev_emerg`].
+    ///
+    /// [`dev_emerg`]: crate::dev_emerg
+    pub fn pr_emerg(&self, args: fmt::Arguments<'_>) {
+        // SAFETY: `klevel` is null-terminated, uses one of the kernel constants.
+        unsafe { self.printk(bindings::KERN_EMERG, args) };
+    }
+
```

Discussion: Discuss about error handling ….

Discussion: Would this be acceptable? With a separate patch introducing the err_info() macro?

```
--- a/samples/rust/rust_minimal.rs
+++ b/samples/rust/rust_minimal.rs
@@ -2,6 +2,7 @@

 //! Rust minimal sample.

+use kernel::err_info;
 use kernel::prelude::*;

 module! {
@@ -22,9 +23,9 @@ fn init(_name: &'static CStr, _module: &'static ThisModule) -> Result<Self> {
         pr_info!("Am I built-in? {}\n", !cfg!(MODULE));

         let mut numbers = Vec::new();
-        numbers.push(72, GFP_KERNEL)?;
-        numbers.push(108, GFP_KERNEL)?;
-        numbers.push(200, GFP_KERNEL)?;
+        numbers.push(72, GFP_KERNEL).inspect_err(|e| err_info!(e))?;
+        numbers.push(108, GFP_KERNEL).inspect_err(|e| err_info!(e))?;
+        numbers.push(200, GFP_KERNEL).inspect_err(|e| err_info!(e))?;

         Ok(RustMinimal { numbers })
     }
```

Note: Many thanks to Miguel for commenting an early version of these slides!

Backup: Example

Alice' binder-rfc:
https://github.com/Darksonn/linux/tree/rust-binder-rfc

~220 places where the ?-operator is used

Are all ok to return/exit 'silently' in case of an error?