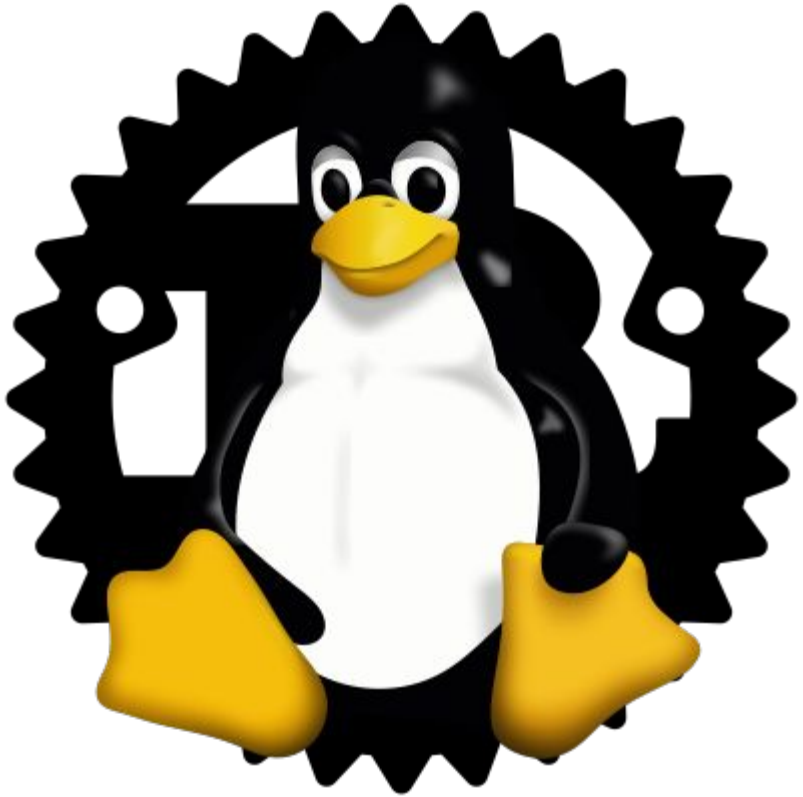


Rust for Linux Networking Tutorial

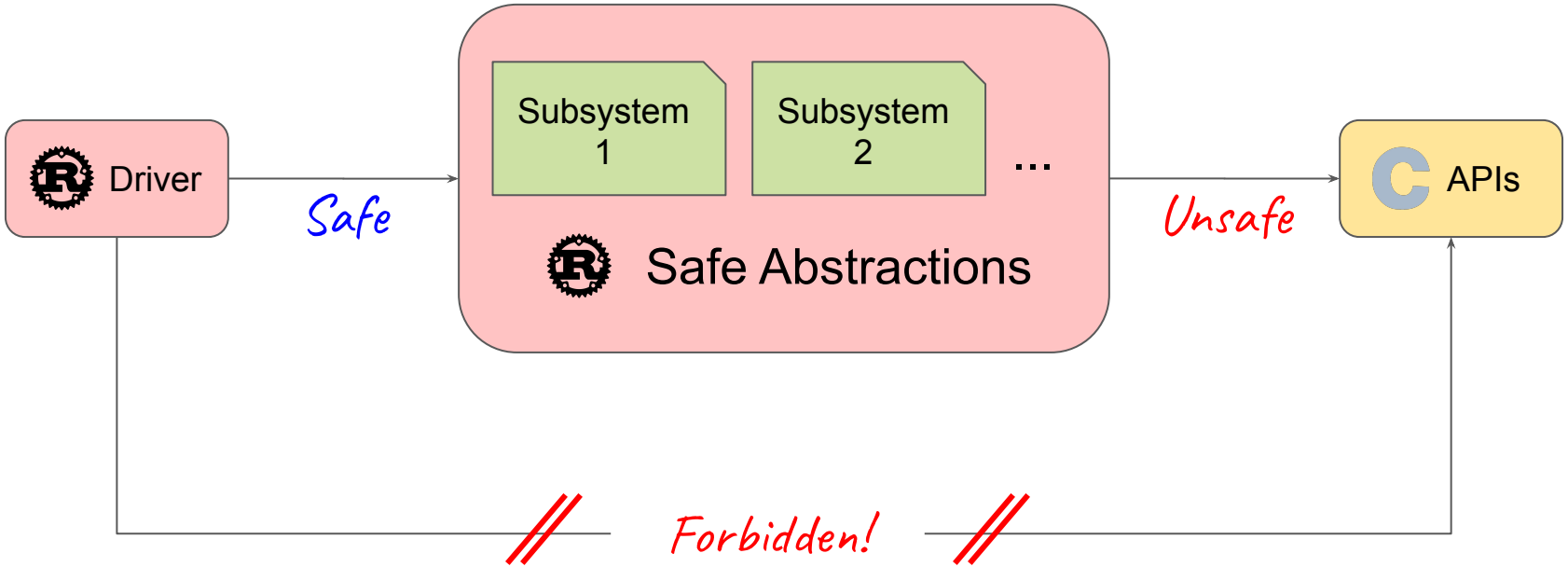
Wedson Almeida Filho
Miguel Ojeda

Agenda

- Key concepts
- Status update
- Writing a synchronous echo server
  - In C and Rust
- Writing an asynchronous echo server
  - In C and Rust
- Async Rust
- Async Rust in the Linux kernel

Key concepts

Encapsulating unsafety



Key concepts

Safe function: a function that does not trigger *Undefined Behavior* in any context and/or for any possible inputs.

Unsafe function: a function that is not safe.

Key concepts

```
int f(int a, int b) {
    return a / b;
}
```

Key concepts

```
int f(int a, int b) {
    return a / b;
}
```

UB ∀x `f(x, 0);`

UB `f(INT_MIN, -1);`

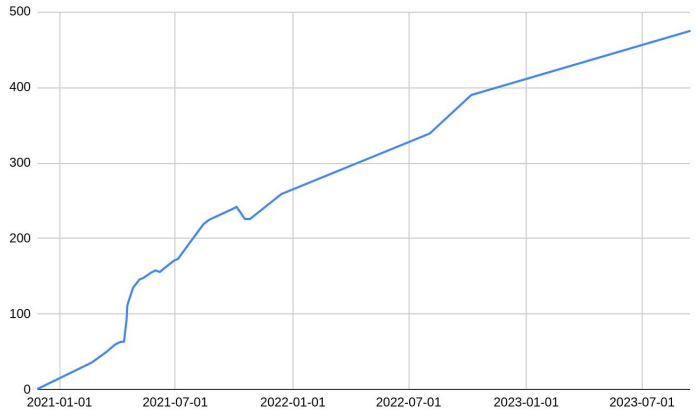
Status update

Growing Community

~460 subscribers in the `rust-for-linux` mailing list.

From ~340 last year.

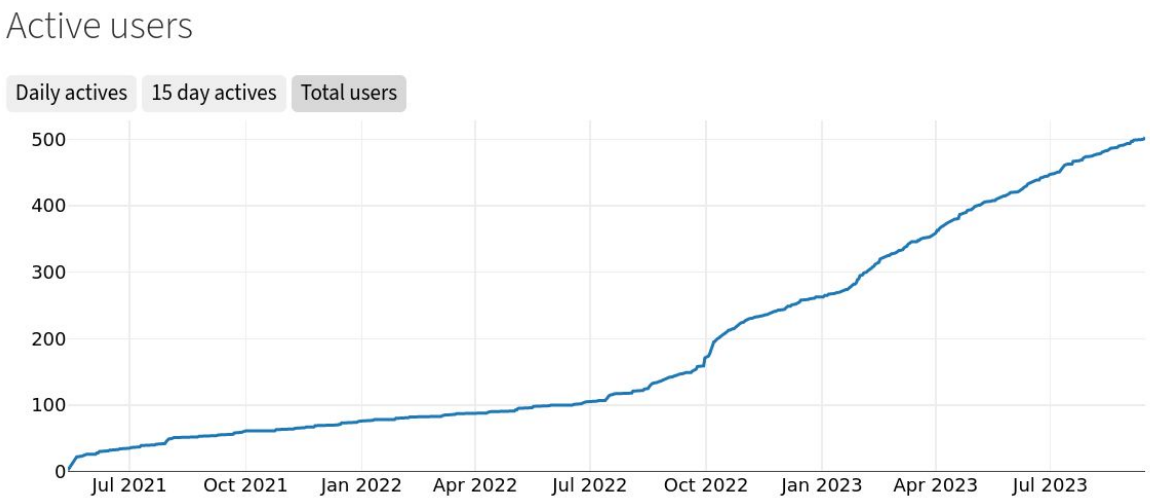
Similar to the BPF and `linux-rt-users` lists.



— https://subspace.kernel.org/vger.kernel.org.html

Growing Community

The Zulip instance (i.e. chat) is growing too: ~530 users now!

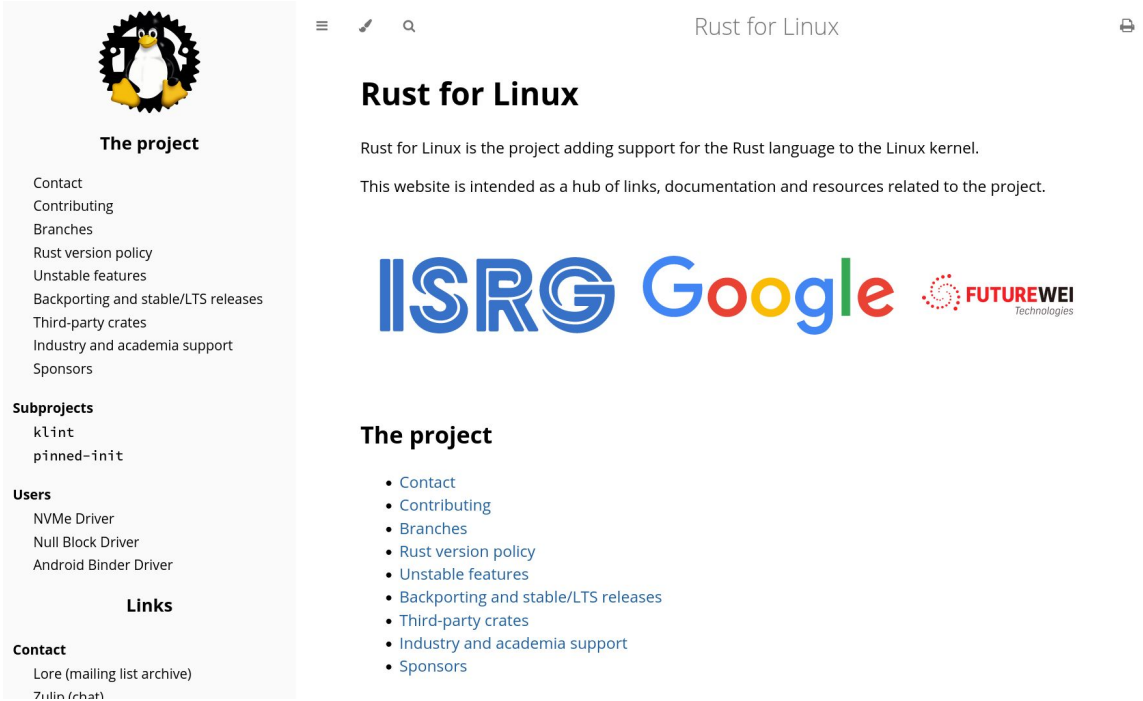


— https://rust-for-linux.zulipchat.com/stats

Growing Core Team

```
RUST
M:    Miguel Ojeda <ojeda@kernel.org>
M:    Alex Gaynor <alex.gaynor@gmail.com>
M:    Wedson Almeida Filho <wedsonaf@gmail.com>
R:    Boqun Feng <boqun.feng@gmail.com>
R:    Gary Guo <gary@garyguo.net>
R:    Björn Roy Baron <bjorn3_gh@protonmail.com>
R:    Benno Lossin <benno.lossin@proton.me>
R:    Andreas Hindborg <a.hindborg@samsung.com>
R:    Alice Ryhl <aliceryhl@google.com>
L:    rust-for-linux@vger.kernel.org
S:    Supported
W:    https://rust-for-linux.com
B:    https://github.com/Rust-for-Linux/linux/issues
C:    zulip://rust-for-linux.zulipchat.com
P:    https://rust-for-linux.com/contributing
T:    git https://github.com/Rust-for-Linux/linux.git rust-next
F:    Documentation/rust/
F:    rust/
F:    samples/rust/
F:    scripts/*rust*
K:    \b(?i:rust)\b
```

rust-for-linux.com



— https://rust-for-linux.com

Sponsors & Industry support

— https://rust-for-linux.com/sponsors
— https://rust-for-linux.com/industry-and-academia-support
— https://www.memorysafety.org/initiative/linux-kernel/

Related projects

`rustc_codegen_gcc` — Antoni Boucher

Compiles & QEMU-boots mainline without source changes.

https://github.com/rust-lang/rustc_codegen_gcc

GCC Rust (`gccrs`) — Arthur Cohen, Philip Herron

Upstreaming started in GCC 13.1, planned initial release for 14.1.

https://github.com/Rust-GCC/gccrs

Coccinelle for Rust — Julia Lawall, Tathagata Roy

Recently published.

https://gitlab.inria.fr/coccinelle/coccinelleforrust

Upstreamed code & RFCs/WIP

6.1: Initial merge (minimal support, Rust 1.62.0).

6.2: `Opaque`, `Either`, `CString`, `CStr`, `BStr`, `#[vtable]`, `concat_idents!`, `{static,build}_assert!`, the rest of `pr_*!` and more error codes, `dbg!`...

6.3: `Arc`, `ArcBorrow`, `UniqueArc`, `ForeignOwnable`, `ScopeGuard`.

6.4: pinned-init API, `AlwaysRefCounted`, `ARef`, `Lock`, `Guard`, `Mutex`, `CondVar`, `Task`, `uapi` crate...

6.5: Rust 1.68.2 (first upgrade), pinned-init improvements, `Error`'s `name()` support, `AsRef` for `Arc`...

6.6: Rust documentation tests as KUnit tests, pinned-init features, `paste!`, Rust 1.71.1, bindgen 0.65.1, `rust_is_available` series...

6.7: Workqueue abstractions, Rust 1.73.0, toybox support (Android), x86 IBT, webpage and Maintainer Entry Profile document.

RFCs/WIP: Binder, NVMe, DRM (Apple GPU), VFS (tarfs, PuzzleFS), PHY, mitigations...

Networking

Networking status

- Chicken and egg problem
  - No networking abstractions without users
  - Users wait for networking abstractions to become available
- Some patches in the `rust` branch
- Waiting for actual users
- Kernel for this tutorial: link

Echo server

- Trivial server:
  - Reads data from the peer
  - Writes the same data back to peer
- Has a lot of requirements that are common to servers
- Allows us to not get lost in irrelevant details
- We'll start with a simple synchronous example
  - But doesn't scale well
- We'll migrate to an asynchronous example that scales well
  - But isn't as simple (in C)
- Code for this tutorial: link

Step 1: empty unloadable module

```c
#include <linux/module.h>

static int __init echo_init(void)
{
        return 0;
}

static void __exit echo_exit(void)
{
}

module_init(echo_init);
module_exit(echo_exit);

MODULE_LICENSE("GPL");
```

```rust
//! Rust echo server.
use kernel::prelude::*;

struct EchoServer;
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        Ok(Self)
    }
}

module! {
    type: EchoServer,
    name: "rust_echo_server",
    license: "GPL",
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Thread has a single untyped (`void *`) argument.

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Printf-style formatting of thread name.

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Explicit error handling.

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Mixed pointers and error codes.

```
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Need to increment the refcount on the task. (Without it, count may go to zero if the thread runs to completion.)

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Need to explicitly stop the thread. Otherwise the kernel will crash when trying to run unloaded code.

```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 2: create thread to listen for connections

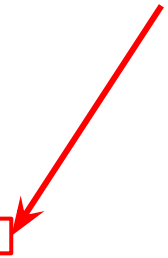
```c
static int __init echo_init(void)
{
        struct task_struct *t =
             kthread_create(echo_listener, NULL,
                    "listener");
        if (IS_ERR(t))
                return PTR_ERR(t);

        listener_thread = t;
        get_task_struct(t);
        wake_up_process(t);

        return 0;
}

static void __exit echo_exit(void)
{
        kthread_stop(listener_thread);
        put_task_struct(listener_thread);
}
```

Need to decrement the refcount. Otherwise we leak the task.



```rust
struct EchoServer(KTask);

impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener()
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

Endianness conversion in host-only code.

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

Explicit cast to generic type.

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

Socket loses its type. It's converted to generic `void *`.

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

Explicit error paths with `goto` statements.

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 3: create listening socket

```c
static int __init echo_init(void)
{
        struct socket *sock;
        struct sockaddr_in addr;
        struct task_struct *t;

        ret = sock_create_kern(&init_net, AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
        if (ret)
                return ret;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr = INADDR_ANY;
        ret = kernel_bind(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (ret)
                goto err_sock;

        ret = kernel_listen(sock, SOMAXCONN);
        if (ret)
                goto err_sock;

        t = kthread_create(echo_listener, sock, "listener");
        if (IS_ERR(t)) {
                ret = PTR_ERR(t);
                goto err_sock;
        }

        /* ... (rest of thread init) */
        return 0;

err_sock:
        sock_release(sock);
        return ret;
}
```

Need to remember to update `ret`, otherwise it will seem like this function succeeded.

```rust
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            || echo_listener(listener)
        )?;
        Ok(Self(task))
    }
}
```

Step 4: accept new connections

```c
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

```rust
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| Task::spawn(
               fmt!("handler"), || echo_handler(s)))
            .and_then(|t| Ok(t.detach()));
    }
}
```

Step 4: accept new connections

```c
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

Implicit cast from generic `void *`.

```rust
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| Task::spawn(
               fmt!("handler"), || echo_handler(s)))
            .and_then(|t| Ok(t.detach()));
    }
}
```

Step 4: accept new connections

```c
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

Implicit cast to generic `void *`.

```rust
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| Task::spawn(
               fmt!("handler"), || echo_handler(s)))
            .and_then(|t| Ok(t.detach()));
    }
}
```

Step 4: accept new connections

```
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

Explicit cleanup on failure to start thread.

```
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| Task::spawn(
               fmt!("handler"), || echo_handler(s)))
            .and_then(|t| Ok(t.detach()));
    }
}
```

Step 4: accept new connections

```
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

Explicit cleanup when returning

```
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| Task::spawn(
               fmt!("handler"), || echo_handler(s)))
            .and_then(|t| Ok(t.detach()));
    }
}
```

Alternative step 4: accept new connections

```c
static int echo_listener(void *data)
{
        struct socket *sock = data;
        int ret = 0;

        while (!kthread_should_stop()) {
                struct socket *newsock;
                struct task_struct *t;

                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t))
                        sock_release(newsock);
        }

        sock_release(sock);

        return ret;
}
```

```rust
fn echo_listener(listener: TcpListener) {
    while !Task::should_stop() {
        if let Ok(s) = listener.accept(true) {
            let ret = Task::spawn(
                fmt!("handler"),
                || echo_handler(s)
            );
            if let Ok(task) = ret {
                task.detach();
            }
        }
    }
}
```

Step 5: read data and echo it back

```
static int echo_handler(void *data)
{
        struct socket *sock = data;
        /* ... */

        for (;;) {
                /* ... */
                iov.iov_base = buf;
                iov.iov_len = sizeof(buf);
                ret = kernel_recvmsg(sock, &msg, &iov, 1, sizeof(buf), 0);
                if (ret <= 0)
                        break;

                write_len = ret;
                to_write = buf;
                while (write_len) {
                        memset(&msg, 0, sizeof(msg));
                        iov.iov_base = to_write;
                        iov.iov_len = write_len;
                        ret = kernel_sendmsg(sock, &msg, &iov, 1, write_len);
                        if (ret <= 0)
                                break;

                        write_len -= ret;
                        to_write += ret;
                }
        }

        sock_release(sock);
        return ret;
}
```

```
fn echo_handler(stream: TcpStream) -> Result {
    let mut buf = [0u8; 512];
    loop {
        let n = stream.read(&mut buf, true)?;
        if n == 0 {
            return Ok(());
        }

        let mut to_write = &buf[..n];
        while !to_write.is_empty() {
            let written =
                stream.write(to_write, true)?;
            to_write = &to_write[written..];
        }
    }
}
```

Step 6: prevent module unload

```
static int echo_handler(void *data)
{
        struct socket *sock = data;
        /* ... */
        sock_release(sock);
        module_put_and_kthread_exit(ret);
}
static int echo_listener(void *data)
{
        /* ... */
        while (!kthread_should_stop()) {
               /* ... */
                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                if (!try_module_get(THIS_MODULE)) {
                        sock_release(newsock);
                        continue;
                }

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t)) {
                        module_put(THIS_MODULE);
                        sock_release(newsock);
                }
        }
        /* ... */
}
```

```
fn echo_listener(listener: TcpListener, module: &'static
ThisModule) {
    while !Task::should_stop() {
            // ...
            Task::spawn_with_module(
                module,
                fmt!("handler"),
                || {
                    let _ = echo_handler(s);
                })
            // ...
    }
}

impl kernel::Module for EchoServer {
    fn init(module: &'static ThisModule) -> Result<Self> {
        // ...
        let task = Task::spawn(fmt!("listener"), move ||
echo_listener(listener, module))?;
        Ok(Self(task))
    }
}
```

Step 6: prevent module unload

```c
static int echo_handler(void *data)
{
        struct socket *sock = data;
        /* ... */
        sock_release(sock);
        module_put_and_kthread_exit(ret);
}
static int echo_listener(void *data)
{
        /* ... */
        while (!kthread_should_stop()) {
               /* ... */
                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                if (!try_module_get(THIS_MODULE)) {
                        sock_release(newsock);
                        continue;
                }

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t)) {
                        module_put(THIS_MODULE);
                        sock_release(newsock);
                }
        }
        /* ... */
}
```

Potential foot gun: calling module_put will unload module.

```rust
fn echo_listener(listener: TcpListener, module: &'static
ThisModule) {
    while !Task::should_stop() {
            // ...
            Task::spawn_with_module(
                module,
                fmt!("handler"),
                || {
                    let _ = echo_handler(s);
                })
            // ...
    }
}

impl kernel::Module for EchoServer {
    fn init(module: &'static ThisModule) -> Result<Self> {
        // ...
        let task = Task::spawn(fmt!("listener"), move ||
echo_listener(listener, module))?;
        Ok(Self(task))
    }
}
```

Step 6: prevent module unload

```c
static int echo_handler(void *data)
{
        struct socket *sock = data;
        /* ... */
        sock_release(sock);
        module_put_and_kthread_exit(ret);
}
static int echo_listener(void *data)
{
        /* ... */
        while (!kthread_should_stop()) {
               /* ... */
                ret = kernel_accept(sock, &newsock, 0);
                if (ret)
                        continue;

                if (!try_module_get(THIS_MODULE)) {
                        sock_release(newsock);
                        continue;
                }

                t = kthread_run(echo_handler, newsock, "handler");
                if (IS_ERR(t)) {
                        module_put(THIS_MODULE);
                        sock_release(newsock);
                }
        }
        /* ... */
}
```

Extra manual cleanup.

```rust
fn echo_listener(listener: TcpListener, module: &'static
ThisModule) {
    while !Task::should_stop() {
            // ...
            Task::spawn_with_module(
                module,
                fmt!("handler"),
                || {
                    let _ = echo_handler(s);
                })
            // ...
    }
}

impl kernel::Module for EchoServer {
    fn init(module: &'static ThisModule) -> Result<Self> {
        // ...
        let task = Task::spawn(fmt!("listener"), move ||
echo_listener(listener, module))?;
        Ok(Self(task))
    }
}
```

Properties of this solution

- Simplicity
  - Linear connection handler
  - Code is easy to follow
- Each connection requires a kernel thread
  - Doesn't scale
- Module cannot be unloaded while there are inflight connections
  - Can't stop accepting connections
  - Could be addressed by keeping track of running thread

C Async Server

Properties

- Single thread for listener
  - There are few (1 in our case) threads to accept connections
- Use shared workqueue to perform work
- State machine with two states:
  - Reading from tcp stream
  - Writing (echoing) to tcp stream
- Socket notifications trigger state machine to run
- Explicit tracking of accepted connections
  - And cleanup on unload
- Requires synchronisation between state machine and unload

Representation of a connection

```
struct connection {
        struct socket *sock;
        bool is_reading;
        char *next_write;
        int pending_write;
        struct work_struct work;
        struct wait_queue_entry wq_entry;
        struct list_head links;
        char buf[512];
};
```

State machine

```
void echo_work(struct work_struct *work)
{
        struct connection *conn =
                 container_of(work, struct connection, work);
        struct kvec iov;
        int ret;

        for (;;) {
                struct msghdr msg = {};

                if (conn->is_reading) {
                        iov.iov_base = conn->buf;
                        iov.iov_len = sizeof(conn->buf);
                        ret = kernel_recvmsg(conn->sock, &msg, &iov, 1,
                                        sizeof(conn->buf), MSG_DONTWAIT);
                        if (ret <= 0) {
                                if (ret != -EAGAIN)
                                        cleanup_conn(conn);
                                return;
                        }
                        conn->is_reading = false;
                        conn->pending_write = ret;
                        conn->next_write = conn->buf;
                } else {
                        msg.msg_flags = MSG_DONTWAIT;
                        iov.iov_base = conn->next_write;
                        iov.iov_len = conn->pending_write;
                        ret = kernel_sendmsg(conn->sock,
                              &msg, &iov, 1,
                              conn->pending_write);
                        if (ret <= 0) {
                                if (ret != -EAGAIN)
                                        cleanup_conn(conn);
                                return;
                        }

                        conn->pending_write -= ret;
                        conn->next_write += ret;
                        conn->is_reading =
                              conn->pending_write == 0;
                }
        }
}
```

Getting socket notifications

```
static int wake_callback(struct wait_queue_entry *entry, unsigned mode, int flags, void *key)
{
        struct connection *conn = container_of(entry, struct connection, wq_entry);
        /* TODO: Check mask. */
        queue_work(system_wq, &conn->work);
        return 1;
}
```

Keeping track of connections

```
static LIST_HEAD(connections);
static DEFINE_MUTEX(conn_mutex);
```

Initialiasing a new connection

```
conn = kmalloc(sizeof(*conn), GFP_KERNEL);
if (!conn) {
        sock_release(newsock);
        continue;
}

conn->sock = newsock;
conn->is_reading = true;

INIT_WORK(&conn->work, echo_work);
init_waitqueue_func_entry(&conn->wq_entry, wake_callback);
add_wait_queue(&conn->sock->wq.wait, &conn->wq_entry);

mutex_lock(&conn_mutex);
list_add(&conn->links, &connections);
mutex_unlock(&conn_mutex);

/* Initial iteration. */
queue_work(system_wq, &conn->work);
```

Unloading the module

```
mutex_lock(&conn_mutex);
while (!list_empty(&connections)) {
        struct connection *conn = container_of(connections.next,
                        struct connection, links);
        list_del_init(&conn->links);
        mutex_unlock(&conn_mutex);

        /* Prevent notifications. */
        remove_wait_queue(&conn->sock->wq.wait, &conn->wq_entry);

        /* Cancel pending work and wait for inflight ones to finish. */
        cancel_work_sync(&conn->work);
        sock_release(conn->sock);
        kfree(conn);

        mutex_lock(&conn_mutex);
}
mutex_unlock(&conn_mutex);
```

Cleaning up inflight connections

```
void cleanup_conn(struct connection *conn)
{
        mutex_lock(&conn_mutex);
        if (conn->links.next == &conn->links) {
                /* Has already been removed. */
                mutex_unlock(&conn_mutex);
                return;
        }
        list_del_init(&conn->links);
        mutex_unlock(&conn_mutex);

        /* Prevent notifications. */
        remove_wait_queue(&conn->sock->wq.wait, &conn->wq_entry);

        /* If there's work pending, cancel it. */
        cancel_work(&conn->work);

        sock_release(conn->sock);
        kfree(conn);
}
```

Rust Async Server

State machine

```
fn echo_handler(stream: TcpStream) -> Result {
    let mut buf = [0u8; 512];
    loop {
        let n = stream.read(&mut buf, true)?;
        if n == 0 {
            return Ok(());
        }

        let mut to_write = &buf[..n];
        while !to_write.is_empty() {
            let written =
                stream.write(to_write, true)?;
            to_write = &to_write[written..];
        }
    }
}
```

```
async fn echo_handler(stream: TcpStream) -> Result {
    let mut buf = [0u8; 512];
    loop {
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            return Ok(());
        }

        let mut to_write = &buf[..n];
        while !to_write.is_empty() {
            let written =
                stream.write(to_write).await?;
            to_write = &to_write[written..];
        }
    }
}
```

Listener loop

```
fn echo_listener(
    listener: TcpListener,
    module: &'static ThisModule,
) {
    while !Task::should_stop() {
        let _ = listener
            .accept(true)
            .and_then(|s| {
                Task::spawn_with_module(module,
fmt!("handler"), || {
                    let _ = echo_handler(s);
                })
            })
            .and_then(|t| Ok(t.detach()));
    }
}
```

```
async fn echo_listener(
    listener: TcpListener,
    executor: Arc<Executor>,
) {
    loop {
        let _ = listener
            .accept()
            .await
            .and_then(|s|
               spawn_task!(executor.as_arc_borrow(),
                 echo_handler(s)));
    }
}
```

Module initialisation

```
struct EchoServer(KTask);
impl kernel::Module for EchoServer {
    fn init(
        module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let task = Task::spawn(
            fmt!("listener"),
            move || echo_listener(listener, module)
        )?;
        Ok(Self(task))
    }
}
```

```
struct EchoServer(AutoStopHandle<Executor>);
impl kernel::Module for EchoServer {
    fn init(
        _module: &'static ThisModule
    ) -> Result<Self> {
        let addr = SocketAddr::V4(
            SocketAddrV4::new(Ipv4Addr::ANY, 8080));
        let listener = TcpListener::try_new(
            net::init_ns(), &addr)?;
        let handle = Executor::try_new(
            kernel::workqueue::system())?;
        spawn_task!(
            handle.executor(),
            echo_listener(
                listener, handle.executor().into())
        )?;
        Ok(Self(handle))
    }
}
```

Async Rust

How does it work?

- We talked about it at OSS North America last year: link

- In summary:
  - Compiler automatically creates a state machine from thread-like code
  - Kernel crate implements executors and reactors

Workqueue Executor

Spawning tasks

- Allocates task: contains future plus executor-specific state (e.g., `work_struct`)
- Adds to task list
- Wakes task up

Waking tasks up

- Enqueues task for running (e.g., `queue_work_on`)
- On worker thread: accesses revocable task, poll future, cleans it up when it completes

Tearing down

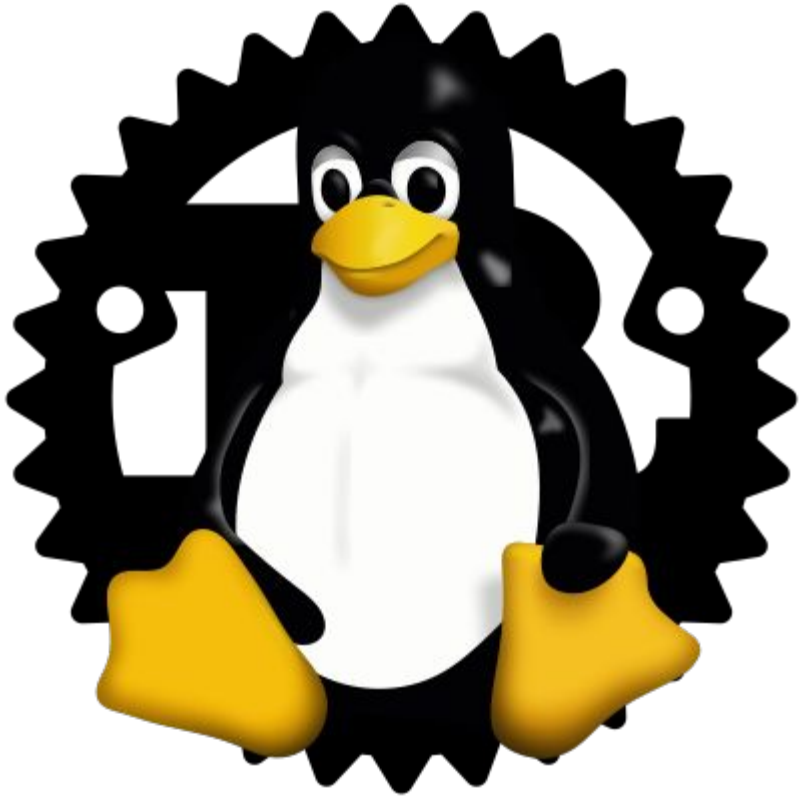
- All state is dropped

Socket Reactor

- Initialisation
  - Pinned larger struct containing some state plus wait queue entry (`wait_queue_entry`)
  - Wait queue entry with a custom function (`init_waitqueue_func_entry`)
  - Adds entry to the socket's wait queue (`add_wait_queue`)
- Waking up
  - Wait queue callback is called: uses `container_of` to get to outer struct
  - Checks mask for filter callbacks (`EPOLLIN`, `EPOLLOUT`, etc)
  - Calls `Waker::wake` to instruct executor to run task again
- Cleaning up
  - Removes entry from socket wait queue (`remove_wait_queue`)

Thank you!

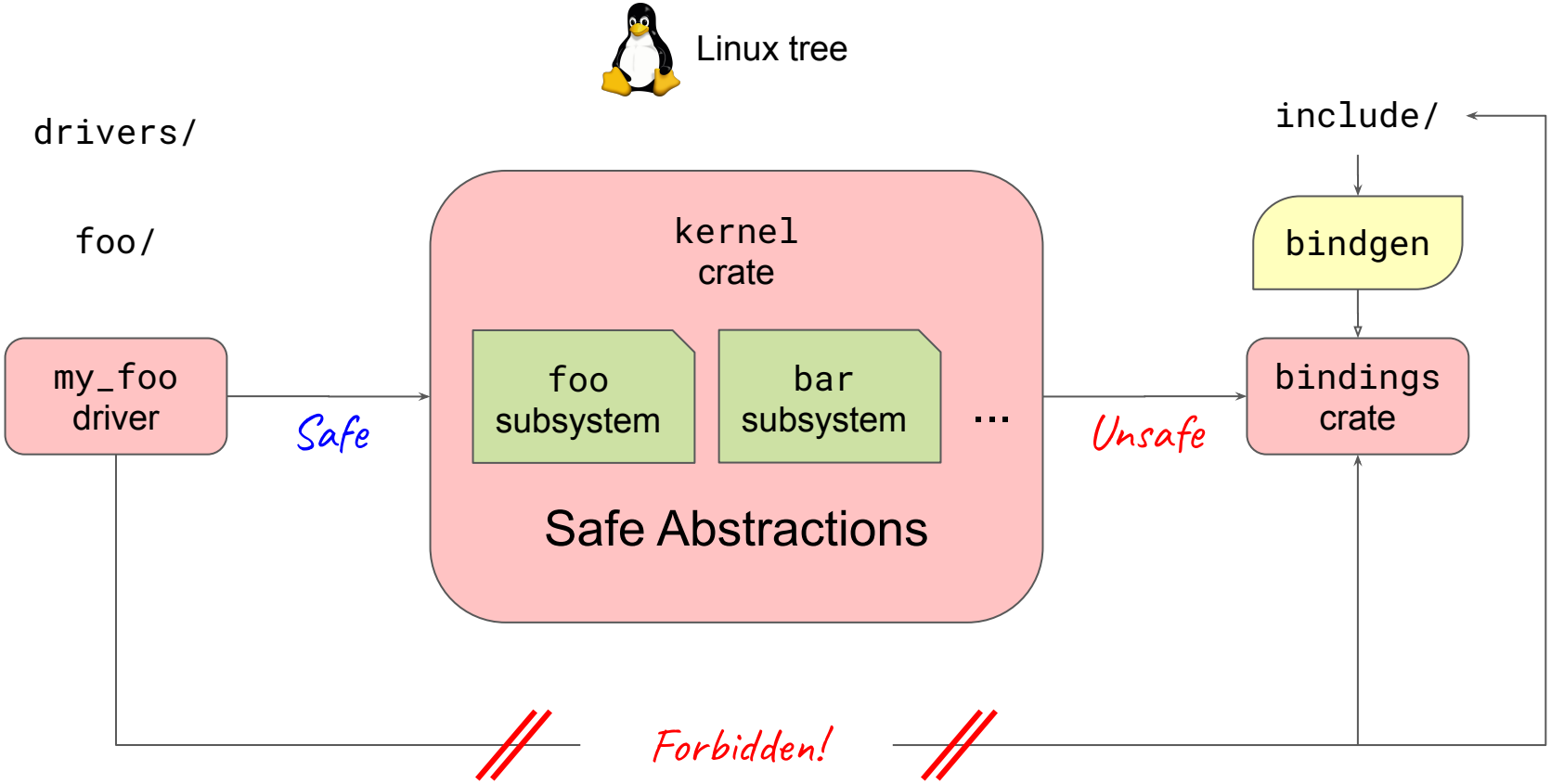
Questions?

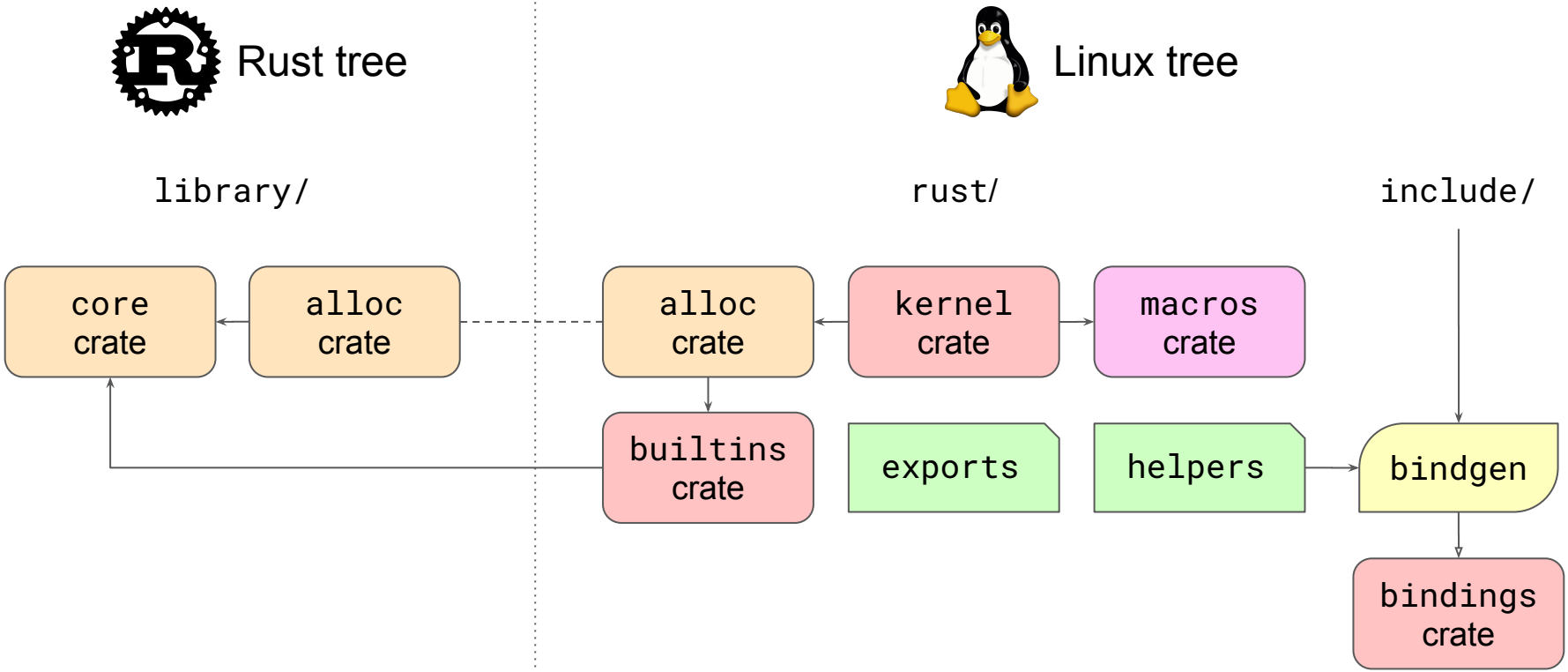


Rust for Linux Networking Tutorial

Wedson Almeida Filho
Miguel Ojeda

Backup slides


Linux tree
drivers/
foo/
my_foo
driver
Safe
kernel
crate
foo
subsystem
bar
subsystem
...
Safe Abstractions
Unsafe
include/
bindgen
bindings
crate
Forbidden!


Rust tree
Linux tree
library/
rust/
include/
core crate
alloc crate
alloc crate
kernel crate
macros crate
builtins crate
exports
helpers
bindgen
bindings crate

Key concepts

```
int f(int a, int b) {
    if (b == 0)
        abort();

    if (a == INT_MIN && b == -1)
        abort();

    return a / b;
}
```