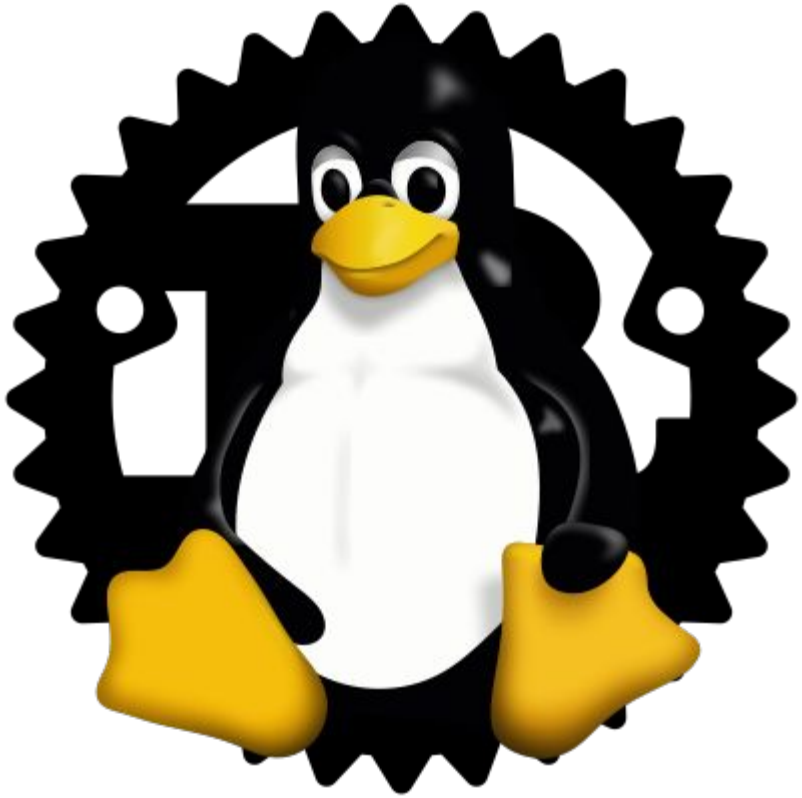


Rust for Linux
Status Update

Miguel Ojeda
Wedson Almeida Filho

Rust for Linux

The project aims to bring Rust support to the Linux kernel as a first-class language.

This includes providing support for writing kernel modules in Rust, such as drivers or filesystems, with as little unsafe code as possible (potentially none).

The last year – Infrastructure

- Removed panicking allocations.
- Moved to Edition 2021 of the Rust language.
- Moved to stable releases of the Rust compiler.
  - And started to track the latest version.
- More architectures initial support (e.g. `arm` (32-bit) and `riscv`).
- Testing support.
  - Including running documentation tests inside the kernel as KUnit tests.
- Support for “hostprogs” written in Rust.
- On-the-fly generation of target specification files based on the kernel configuration.

The last year – Abstractions

PrimeCell PL061 GPIO example driver.

Functionality for platform and AMBA drivers, red-black trees, file descriptors, efficient bit iterators, tasks, files, IO vectors, power management callbacks, IO memory, IRQ chips, credentials, VMA, Hardware Random Number Generators, networking...

Synchronization features such as RW semaphores, revocable mutexes, raw spinlocks, a no-wait lock, sequence locks...

Replaced `Arc` and `Rc` from the `alloc` crate with a simplified kernel-based `Ref`.

Better panic diagnostics and simplified pointer wrappers.

The beginning of Rust `async` support.

The last year – Related projects

Rust stabilized a few unstable features we used.

Improvements on the Rust compiler, standard library and tooling.

> e.g. `rustc_parse_format` compile on stable, the addition of the `no_global_oom_handling` and `no_fp_fmt_parse` gates...

`binutils`/`gdb`/`libiberty` got support for Rust v0 demangling.

Intel's 0Day/LKP kernel test robot started testing a build with Rust support enabled.

Linaro's TuxSuite added Rust support.

`rustc_codegen_gcc` (the `rustc` backend for GCC) and GCC Rust (a Rust frontend for GCC) saw a lot of progress.

Compiler Explorer added the alternative compilers for Rust, as well as other features such as MIR and macro expansion views.

Blog post

The origins, progress and future of Rust for Linux.

https://www.memorysafety.org/blog/memory-safety-in-linux-kernel/

ISRG

OSSNA 2022

More details on the LinuxCon session:

https://www.youtube.com/watch?v=jIX2gYsgr10

Kangrejos

The Rust for Linux Workshop

An event where people involved in the Rust for Linux discussions can meet in a single place just before LPC.

https://kangrejos.com

https://lwn.net/Archives/ConferenceIndex/#Kangrejos

Google ISRG

v9 — Trimming down the patch series

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).
- 60% of the `alloc` crate (the “adapt” commit is only 100 lines).

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).
- 60% of the `alloc` crate (the “adapt” commit is only 100 lines).
- From 40 to 13 klines.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).
- 60% of the `alloc` crate (the “adapt” commit is only 100 lines).
- From 40 to 13 klines.
- Could be made even more minimal.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).
- 60% of the `alloc` crate (the “adapt” commit is only 100 lines).
- From 40 to 13 klines.
- Could be made even more minimal.

The goal is to get the “core” Rust support in first, then start upstreaming the rest piece by piece.

v9 — Trimming down the patch series

v8 was the last “full” version of the patch series.

v9 is a trimmed down v8:

- Enough support to compile a minimal Rust kernel module.
- Includes sample that uses `Vec<i32>` and `pr_info!` macro.
- 3% of the `kernel` crate (500 lines).
- 60% of the `alloc` crate (the “adapt” commit is only 100 lines).
- From 40 to 13 klines.
- Could be made even more minimal.

The goal is to get the “core” Rust support in first, then start upstreaming the rest piece by piece.

The full repository will continue to be available at https://github.com/Rust-for-Linux/linux.

v8 — Limited file system support

Introduction of several Rust wrappers

SuperBlock, INode, Dentry, Filename, Type, Context, Registration.

`module_fs` macro

Simplified definition of modules that only define a file system.

Support for file system parameters

Flags, booleans, strings, enums, u32s (dec, hex, oct), u64s.

But file system must be empty

More on this later.

Support for work queues: fallible enqueuing

```
spawn_work_item!(workqueue::system(), || pr_info!("Hello from a work item\n"))?;
```

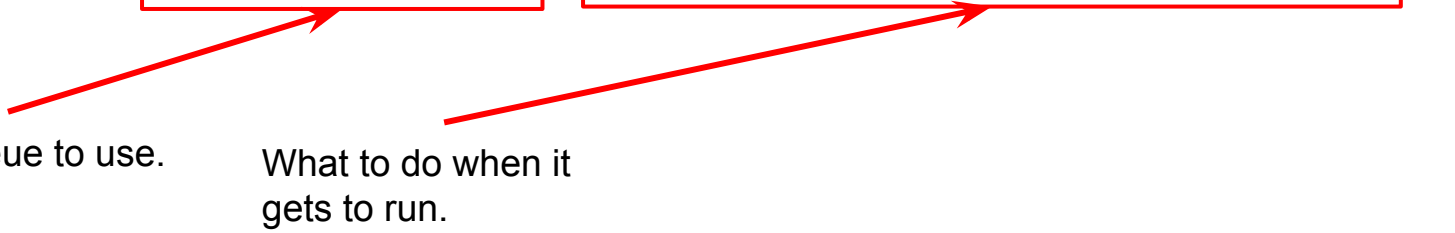
Support for work queues: fallible enqueuing

```
spawn_work_item!(workqueue::system(), || pr_info!("Hello from a work item\n"))?;
```

Which queue to use.

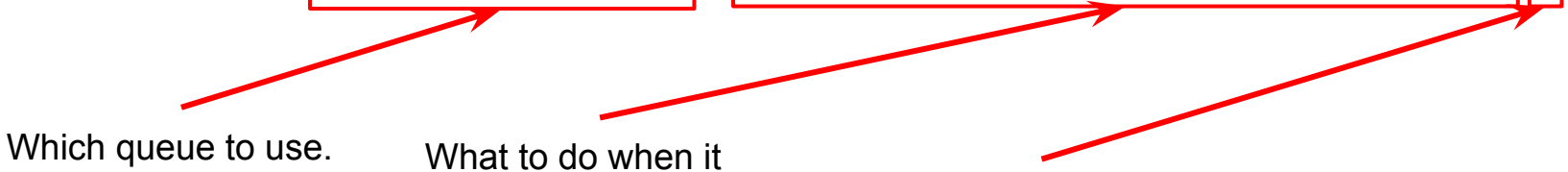
Support for work queues: fallible enqueuing

```
spawn_work_item!(workqueue::system(), || pr_info!("Hello from a work item\n"))?;
```



Support for work queues: fallible enqueuing

```
spawn_work_item!(workqueue::system(), || pr_info!("Hello from a work item\n"))?;
```

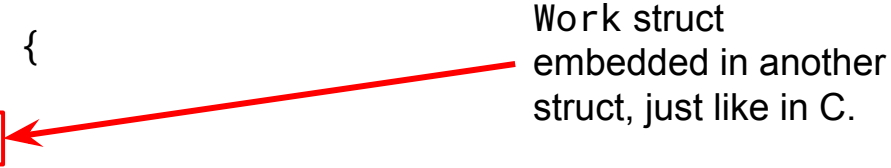


Support for work queues: infallible enqueuing

```
struct Example {
    // [...]
    work: Work,
    // [...]
}

kernel::impl_self_work_adapter!(Example, work, |e| {
    // Do work.
});

fn example(e: RefBorrow<'_, Example>) {
    // [...]
    workqueue::system().enqueue(e.into());
    // [...]
}
```

Support for work queues: infallible enqueuing

```
struct Example {
    // [...]
    work: Work,
    // [...]
}

kernel::impl_self_work_adapter!(Example, work, |e| {
    // Do work.
});

fn example(e: RefBorrow<'_, Example>) {
    // [...]
    workqueue::system().enqueue(e.into());
    // [...]
}
```



Support for work queues: infallible enqueuing

```
struct Example {
    // [...]
    work: Work,
    // [...]
}

kernel::impl_self_work_adapter!(Example, work, |e| {
    // Do work.
});

fn example(e: RefBorrow<'_, Example>) {
    // [...]
    workqueue::system().enqueue(e.into());
    // [...]
}
```

`Work` struct embedded in another struct, just like in C.

Declare function to run when the work item runs.

Support for work queues: infallible enqueuing

```
struct Example {
    // [...]
    work: Work,
    // [...]
}

kernel::impl_self_work_adapter!(Example, work, |e| {
    // Do work.
});

fn example(e: RefBorrow<'_, Example>) {
    // [...]
    workqueue::system().enqueue(e.into());
    // [...]
}
```

Work struct embedded in another struct, just like in C.

Declare function to run when the work item runs.

Enqueueing never fails, like in C.

Workqueue-based async executor

```
let mut handle = Executor::try_new(workqueue::system())?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;
```

Workqueue-based async executor

```
let mut handle = Executor::try_new(workqueue::system())?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;
```

Creates executor handle that uses the `system` work queue.

Workqueue-based async executor

```
let mut handle = Executor::try_new(workqueue::system())?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;
```

Creates executor handle that uses the `system` work queue.

Creates new task to run in executor.

Workqueue-based async executor

```
let mut handle = Executor::try_new(workqueue::system())?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;
```

Creates executor handle that uses the `system` work queue.

Creates new task to run in executor.

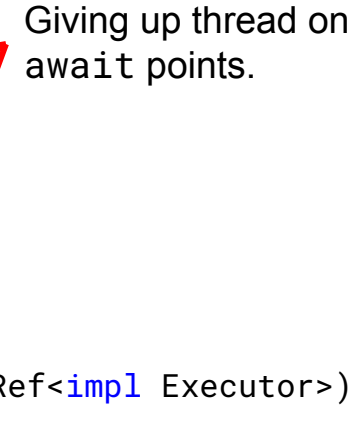
Executor automatically stopped when handle goes out of scope.

Echo server sample

```
async fn echo_server(stream: TcpStream) -> Result {
    let mut buf = [0u8; 1024];
    loop {
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            return Ok(());
        }
        stream.write_all(&buf[..n]).await?;
    }
}

async fn accept_loop(listener: TcpListener, executor: Ref<impl Executor>) {
    loop {
        if let Ok(stream) = listener.accept().await {
            let _ = spawn_task!(executor.as_ref_borrow(), echo_server(stream));
        }
    }
}
```

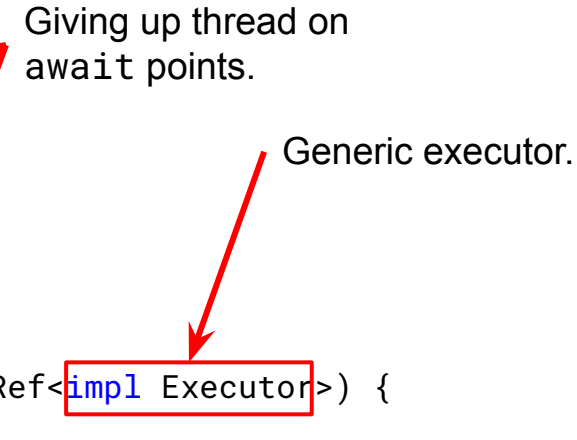
Echo server sample

```
async fn echo_server(stream: TcpStream) -> Result {
    let mut buf = [0u8; 1024];
    loop {
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            return Ok(());
        }
        stream.write_all(&buf[..n]).await?;
    }
}

async fn accept_loop(listener: TcpListener, executor: Ref<impl Executor>) {
    loop {
        if let Ok(stream) = listener.accept().await {
            let _ = spawn_task!(executor.as_ref_borrow(), echo_server(stream));
        }
    }
}
```



Echo server sample

```
async fn echo_server(stream: TcpStream) -> Result {
    let mut buf = [0u8; 1024];
    loop {
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            return Ok(());
        }
        stream.write_all(&buf[..n]).await?;
    }
}

async fn accept_loop(listener: TcpListener, executor: Ref<impl Executor>) {
    loop {
        if let Ok(stream) = listener.accept().await {
            let _ = spawn_task!(executor.as_ref_borrow(), echo_server(stream));
        }
    }
}
```



Echo server sample

```
async fn echo_server(stream: TcpStream) -> Result {
    let mut buf = [0u8; 1024];
    loop {
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            return Ok(());
        }
        stream.write_all(&buf[..n]).await?;
    }
}

async fn accept_loop(listener: TcpListener, executor: Ref<impl Executor>) {
    loop {
        if let Ok(stream) = listener.accept().await {
            let _ = spawn_task!(executor.as_ref_borrow(), echo_server(stream));
        }
    }
}
```

Giving up thread on `await` points.

Generic executor.

New task per connection.

Basic RCU read-side locking

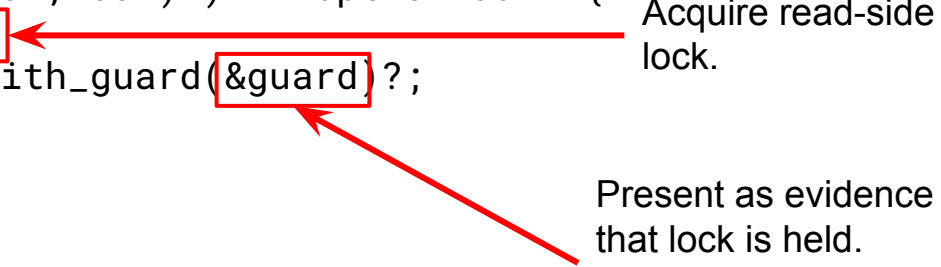
```
fn add_pair(value: &Revocable<(u32, u32)>) -> Option<u32> {
    let guard = rcu::read_lock();
    let pair = value.try_access_with_guard(&guard)?;
    Some(pair.0 + pair.1)
}
```

Basic RCU read-side locking

```
fn add_pair(value: &Revocable<(u32, u32)>) -> Option<u32> {
    let guard = rcu::read_lock();
    let pair = value.try_access_with_guard(&guard)?;
    Some(pair.0 + pair.1)
}
```

Basic RCU read-side locking

```
fn add_pair(value: &Revocable<(u32, u32)>) -> Option<u32> {
    let guard = rcu::read_lock();
    let pair = value.try_access_with_guard(&guard)?;
    Some(pair.0 + pair.1)
}
```

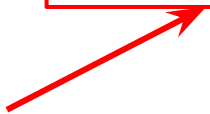


Simple creation of kernel threads

```
Task::spawn(fmt!("task{i}"), || pr_info!("Hello from thread\n"))?;
```

Simple creation of kernel threads

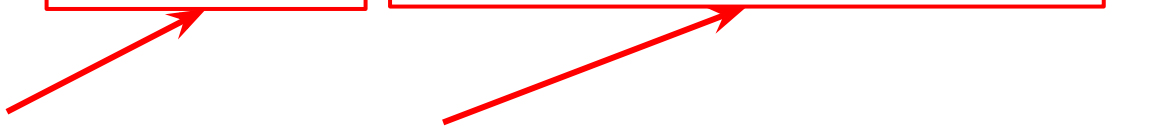
```
Task::spawn(fmt!("task{i}"), || pr_info!("Hello from thread\n"))?;
```



Formatted name of
the task.

Simple creation of kernel threads

```
Task::spawn(fmt!("task{i}"), || pr_info!("Hello from thread\n"))?;
```

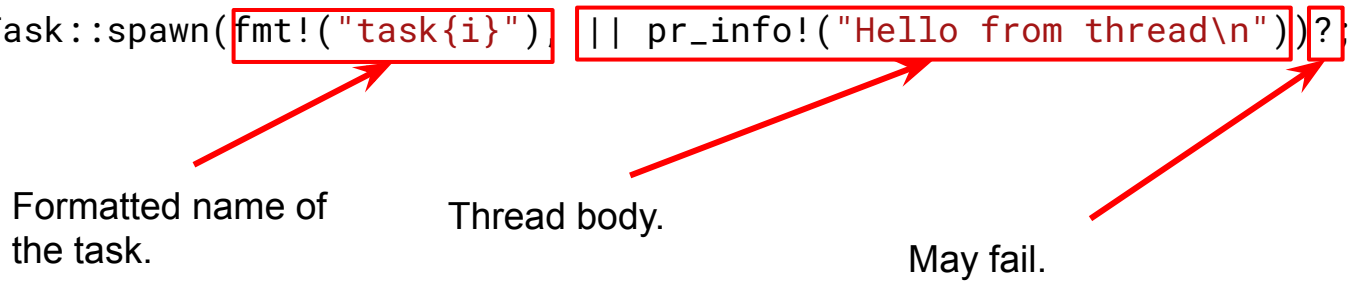


Formatted name of the task.

Thread body.

Simple creation of kernel threads

```
Task::spawn(fmt!("task{i}"), || pr_info!("Hello from thread\n"))?;
```



Miscellaneous

IRQ handling
: Allow drivers to handle IRQs, part of NMVe driver.

`AsyncRevocable`
: Allows revocation to happen asynchronously, when last concurrent user finishes.

`StaticRef`
: Allow creating "ref-counted" globals so that they can be used where `Ref` and `RefBorrow` are expected.

`yield_now`
: yield execution of async task.

`unsafe_list::List`
: Intrusive circular doubly-linked list, head is not self-referential.

Upcoming

Consuming `File` objects

```
impl File {
   pub fn open(name: &CStr, flags: u32) -> Result<ARef<Self>>;
   pub fn read(&self, out: &mut [u8], offset: u64) -> Result<u64>;
   pub fn readdir<T: FnMut(&[u8], u64, u64, u32) -> Result<bool>>(
       &self,
       index: u64,
       cb: T,
   ) -> Result;
   pub fn inode(&self) -> &fs::INode;
   pub fn dentry(&self) -> &fs::DEntry;
   pub fn path(&self) -> &fs::Path;
}

impl Path {
   pub fn lookup(&self, name: &[u8], flags: u32) -> Result<Self>;
   pub fn open(&self, flags: u32, cred: &Credential) -> Result<ARef<file::File>>;
}
```

9p server

No unsafe blocks at all.

Uses async support, async networking, and consuming `File` objects.

Exposes a server on port 564.

9p clients (including the Linux kernel one) can mount it.

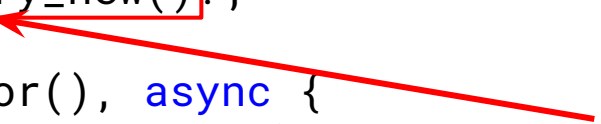
Still WIP, more details here.

Local async executor

```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run(false)?;
```

Local async executor

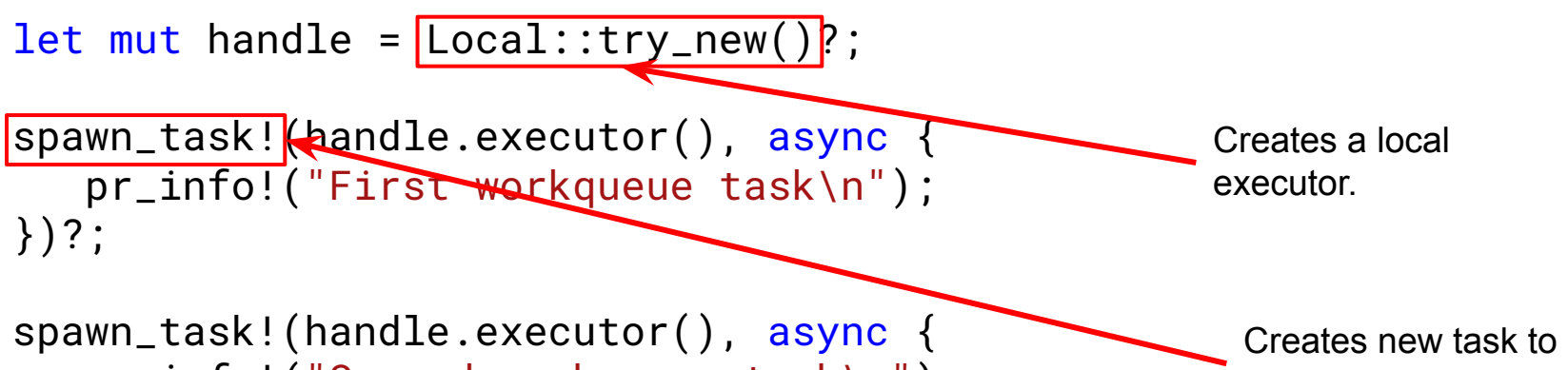
```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run(false)?;
```



Creates a local executor.

Local async executor

```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run(false)?;
```



Local async executor

```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run(false)?;
```

Creates a local executor.

Creates new task to run in executor.

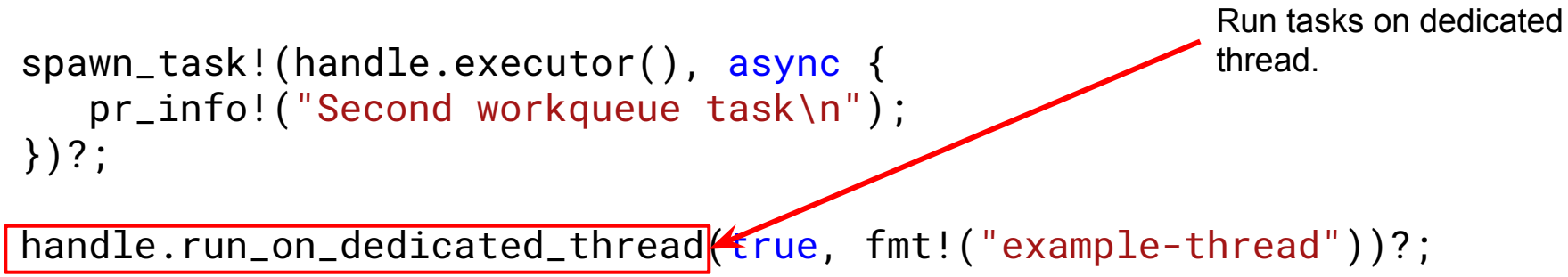
Run tasks on current thread.

Local async executor: dedicated thread

```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run_on_dedicated_thread(true, fmt!("example-thread"))?;
```

Local async executor: dedicated thread

```
let mut handle = Local::try_new()?;

spawn_task!(handle.executor(), async {
    pr_info!("First workqueue task\n");
})?;

spawn_task!(handle.executor(), async {
    pr_info!("Second workqueue task\n");
})?;

handle.run_on_dedicated_thread(true, fmt!("example-thread"))?;
```



Static but non-empty file system support

```
fn fill_super(_data: (), sb: fs::NewSuperBlock<'_, Self>) -> Result<&fs::SuperBlock<Self>> {
    let sb = sb.init(
        (),
        &fs::SuperParams {
            magic: 0x72757374,
            ..fs::SuperParams::DEFAULT
        },
    )?;
    let root = sb.try_new_populated_root_dentry(
        &[],
        kernel::fs_entries![
            file("test2", 0o600, "def\n".as_bytes(), FsFile),
            char("test3", 0o600, [].as_slice(), (10, 125)),
            sock("test4", 0o755, [].as_slice()),
            fifo("test5", 0o755, [].as_slice()),
            block("test6", 0o755, [].as_slice(), (1, 1)),
            dir(
                "dir1",
                0o755,
                [].as_slice(),
                [
                    file("test1", 0o600, "abc\n".as_bytes(), FsFile),
                    file("test2", 0o600, "def\n".as_bytes(), FsFile),
                ]
            ),
        ],
    )?;
    sb.init_root(root)
}
```

Static but non-empty file system support

```
fn fill_super(_data: (), sb: fs::NewSuperBlock<'_, Self>) -> Result<&fs::SuperBlock<Self>> {
    let sb = sb.init(
        (),
        &fs::SuperParams {
            magic: 0x72757374,
            ..fs::SuperParams::DEFAULT
        },
    )?;
    let root = sb.try_new_populated_root_dentry(
        &[],
        kernel::fs_entries![
            file("test2", 0o600, "def\n".as_bytes(), FsFile),
            char("test3", 0o600, [].as_slice(), (10, 125)),
            sock("test4", 0o755, [].as_slice()),
            fifo("test5", 0o755, [].as_slice()),
            block("test6", 0o755, [].as_slice(), (1, 1)),
            dir(
                "dir1",
                0o755,
                [].as_slice(),
                [
                    file("test1", 0o600, "abc\n".as_bytes(), FsFile),
                    file("test2", 0o600, "def\n".as_bytes(), FsFile),
                ]
            ),
        ],
    )?;
    sb.init_root(root)
}
```

Implements `file::Operations`.

Next milestones

More users or use cases inside the kernel, including example drivers.

Extending the current integration of the kernel documentation, testing and other tools.

Getting more subsystem maintainers, companies and researchers involved.

And, of course, getting merged into the mainline kernel!

Events

Kernel Summit at LPC 2022

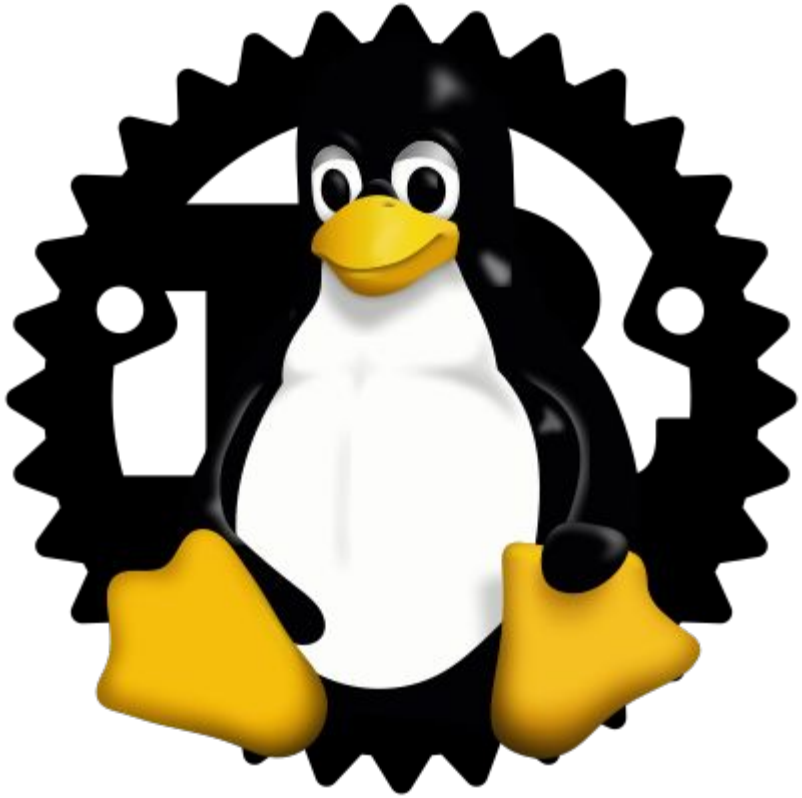
Join us for the Rust session on Wednesday at 15:45.
https://lpc.events/event/16/contributions/1225/

Two more Linux Foundation Live Mentorship Series are coming

https://events.linuxfoundation.org/lf-live-mentorship-series/

Thank you!

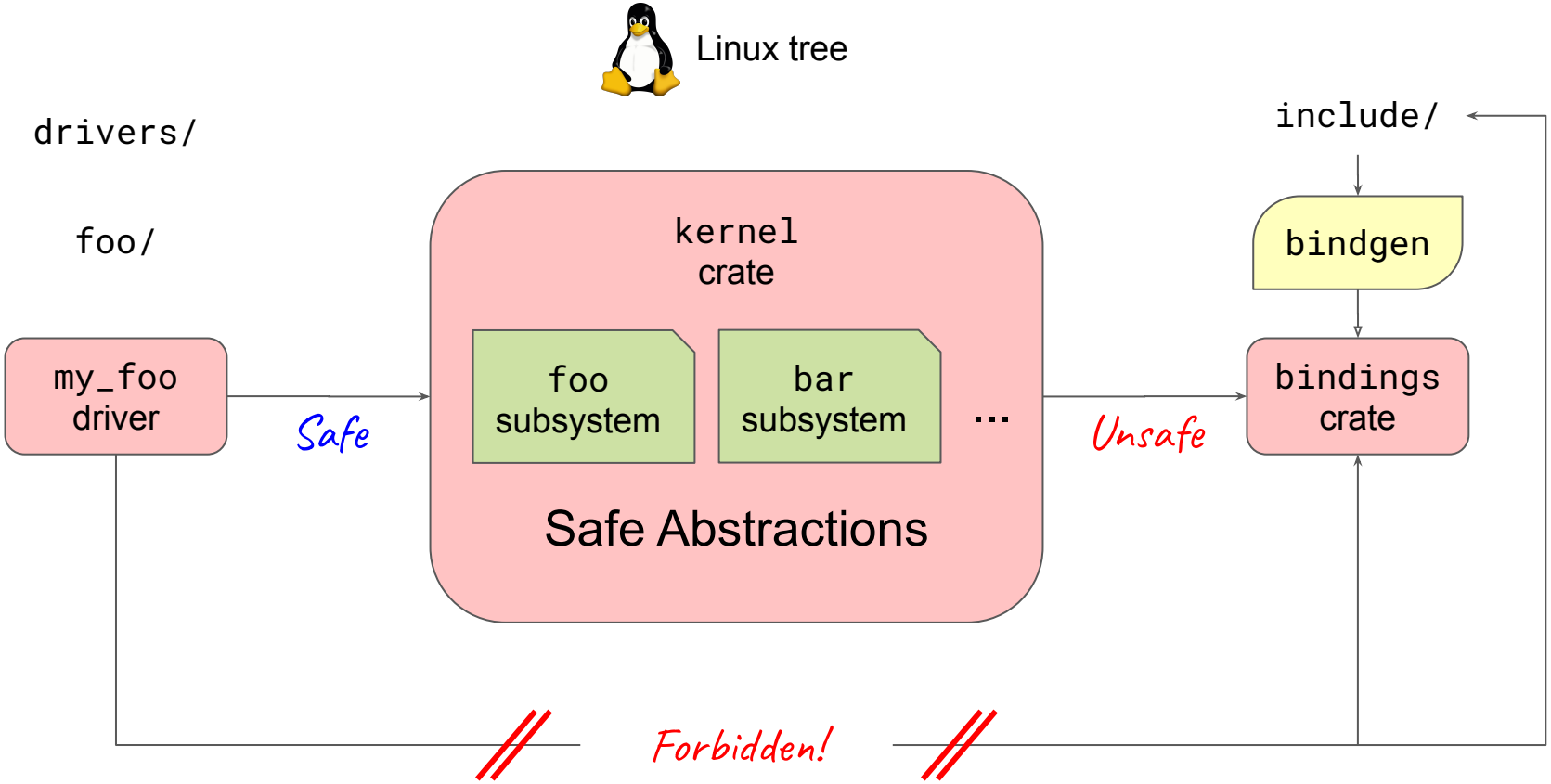
Questions?

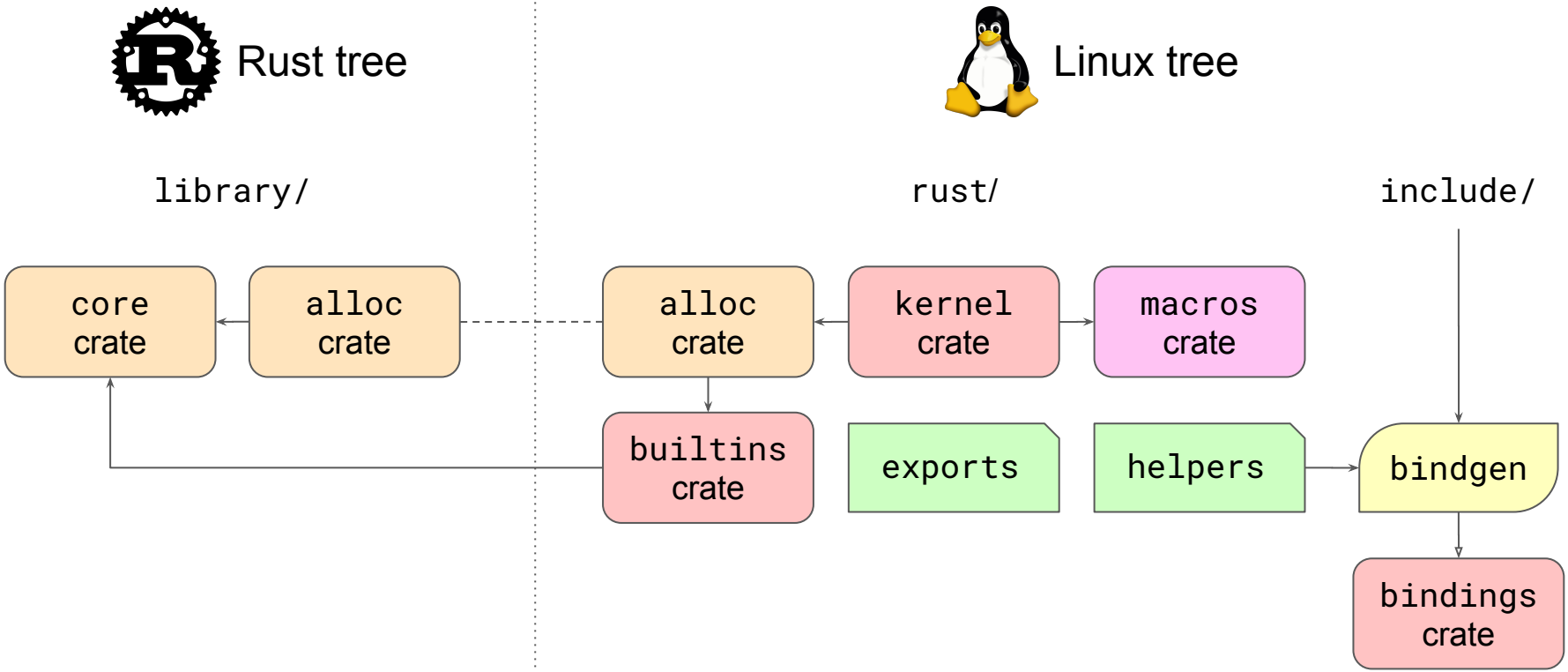


Rust for Linux
Status Update

Miguel Ojeda
Wedson Almeida Filho

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