プログラミング言語 Rust による Linux デバイスドライバの開発

Development of a Linux device driver by the Rust programming language

追川 修一 [1*]
Shuichi Oikawa[1*]

[1] 東京都立産業技術大学院大学 Advanced Institute of Industrial Technology
[*]Corresponding author: Shuichi Oikawa, oikawa-shuichi@aiit.ac.jp

Abstract Device drivers, which are components of the operating system kernel, manage and control hardware devices. There are a wide variety of devices from commodity devices, such as storage and network interfaces, to specialized devices, such as sensors and motors. A specific device driver is developed for each of devices. Therefore, there are a number of devices drivers in the operating system kernel. On the other hand, there is a clear trend that device drivers are the source of operating system failures because of the lower skill of device driver developers and insufficient test and verification. This paper explores the possibility of using the Rust programming language to program Linux device drivers in order to reduce their failures by taking advantages of the language safety features.

Keywords systems software; operating systems; programming languages; Linux; Rust

1 はじめに

オペレーティングシステム (OS: Operating System) カーネルは，直接ハードウェア上で動作し，アプリケーションの実行環境を提供するソフトウェアである．OS カーネルは複数の機能モジュールから構成され，その 1 つであるデバイスドライバは，CPU 以外の主に入出力を行うハードウェアデバイスを直接制御するソフトウェアモジュールである．ハードウェアデバイスをアプリケーションから用いるには，デバイスドライバが必要となる．

ハードウェアデバイスは，ストレージやネットワーク等の汎用的なデバイスから，センサやモータ等の専用に開発されることが多いデバイスまで，多種多様である．これら多種多様なハードウェアデバイスのそれぞれに対し，デバイスドライバは開発されるため，OS カーネルのソースコードに占める割合として多くなる．その一方で，OS 開発スキルの低い技術者による開発や不十分なテスト・検証により，デバイスドライバには欠陥が多い傾向にあることが知られている [1].

特に，組込みシステムでは，1) 独自開発されたハードウェアデバイスを組み込むことが多く，専用に開発されたデバイスドライバを必要とする，2) 長期間の連続動作が求められる一方でオペレータの監視下には無い，という理由から，デバイスドライバには欠陥が多く発生する可能性がより高いにも関わらず，より高い信頼性が求められるという，相反した課題があり，高い信頼性を持つデバイスドライバの実装を可能にする開発方法が必要とされる．そこで本論文では，安全性の高いシステムプログラミング言語である Rust を用いて，Linux カーネルのデバイスドライバを開発する方法について述べる．Linux カーネルはバージョン 6.1 で Rust 言語のサポートをメインラインカーネルに取り込み，Rust によるデバイスドライバを含むカーネルモジュールの開発が進むと考えられる．しかしながら，Rust によるデバイスドライバの開発方法についての情報はまだ限定的であるため，先行的に開発を行い，知見を貯めることには大きな意義がある．そこで，本論文では具体的な例とともに，Rust による Linux カーネルのデバイスドライバの開発方法について述べる．

以下， 2 章では Linux カーネルの Rust サポートの動向について述べる．3 章では開発環境についてまとめ，4 章では Rust 言語による Linux デバイスドライバの開発方法について述べる．5 章で考察を行い，6 章で本論文をまとめる．

2 Linux カーネルの Rust サポートの動向

本章では，Linux カーネルの開発動向について，Rust サポートの状況を中心として述べる．

Linux カーネルの機能モジュールを Rust で開発できるようにするためのサポートは，Rust for Linux [2] で開発されてきた．Rust for Linux は，2021 年 3 月に Linux next に取り込まれ [3]，また同年 4 月には RFC も投稿された [4]．その後，活発に議論および開発が進み，2022 年 12 月に Linux カーネルはバージョン 6.1 で Rust サポートをメインラインに取り込むまでに至った．しかしながら，バージョン 6.1 では Rust でカーネルモジュールを作成できる程度であり，すぐに実用的なカーネルモジュールを開発できる段階ではなかった．

本論文執筆時点での最新バージョンは 6.5 である．バージョン 6.2 から 6.5 までの各バージョンで Rust サポートはアップデートされており，構造体やトレイト等の型の定義が順次追加されている．しかしながら，Rust でのデバイスドライバの開発までサポートしている Rust for Linux と比較すると未だ機能的には追いついていないのが現状である．

3 開発環境

本章では，Rust モジュールを含む Linux カーネルの開発環境についてまとめる．Rust コンパイラのインストールおよび Linux カーネルのビルドについては，すでに述べた [5]．ここでは，MacOS をホスト環境として Linux カーネルの開発環境を構築する方法について述べる．

Linux カーネルをビルドするためには，コンパイラ等の開発ツール群の実行環境として Linux が必要である．そこで，開発環境として Linux を用いるのが最も簡単な方法となる．しかしながら，開発以外の作業環境として Linux は最適とは言えない，また用いることができない場合も未だある．例えば，現在の作業環境としての PC は，CPU として Apple M1 Pro を搭載する MacBook Pro である．この PC で Linux は動作するようになってきているものの，まだ試験的である．

Received: 2023-10-29; Published online: 2024-02-29

MacOS では Docker を用いて Linux を実行可能であるため，コンパイラ等の開発ツール群の実行環境として用いることができる．しかしながら，Docker 上の Linux を開発環境として用いるには，以下の 2 つの問題があった．まず，開発対象である Linux カーネルのソースコードの保存である．Docker では，イメージ内での変更は，明示的にイメージ全体を保存しなければ，そのイメージの実行が終了した時点で破棄される．またイメージは，レイヤ構造として保存されていくため，バイナリが生成される開発環境で使用すると大量の差分が作成され，蓄積されてしまうため，開発環境には不向きである．また，Docker 上の Linux 環境は基本的には CLI になる．Linux の GUI は X Window System 上に構築されているため，ホスト環境の MacOS で X Window サーバを実行することで，Linux の GUI を使用できないことはない．しかしながら，MacOS の GUI よりは見劣りしてしまうことが問題である．

上記の 2 つの問題は以下のように解決することができる．まず，Linux カーネルのソースコードの保存については，Docker で永続ボリュームを作成することで解決できる．永続ボリュームは，その名の通り永続的なストレージとして用いることができる Docker イメージからは独立したストレージ領域であり，Docker イメージの実行時にマウント先のディレクトリを指定して用いることができる．そのため，Docker イメージを再起動した後も，または再構築された別のイメージからも利用することができる．永続的にファイルを保存する別の方法としては，ホスト環境のファイルシステムを用いる方法もある．ホスト環境が Linux の場合は，この方法で問題は無い．しかしながらホスト環境が MacOS の場合は，Docker 上の Linux 環境とホスト環境の MacOS のファイルシステムが異なることから生じるセマンティクス上の問題があり，またアクセスのためのオーバヘッドも大きいことから，この方法は機能しないことが判明した．また，ホスト環境の GUI を使用するエディタから Docker イメージ内のファイルを編集する問題は，エディタとして Visual Studio Code（VSCode）を Dev Containers 機能拡張と共に用いることで解決できる．この組み合わせにより，ホスト環境で実行している VSCode は，編集対象とする Docker イメージ内のファイルを，あたかもホスト環境にあるかのように扱うことができる．

以上のように，Docker の永続ボリューム，VSCode, Dev Containers 機能拡張を組み合わせることで，MacOS 上に Linux カーネルの開発環境を構築することができる．

4 Rust 言語による Linux デバイスドライバの開発

本章では，Rust 言語による Linux デバイスドライバの開発について述べる．まず，Rust による最も単純なデバイスドライバモジュールの定義方法について述べる．次に，このデバイスドライバモジュールに FIFO 機能，即ち書き込んだデータを書き込んだ順番で読み出すことのできる機能の実装について述べる．

デバイスドライバモジュールの定義

Rust for Linux におけるデバイスドライバモジュールの定義について述べる．図 1 に，最も単純なデバイスドライバモジュールを定義した例を示す．このモジュールは，デバイスファイルとして /dev/rust_fifo を作成し，そのデバイスファイルがオープンされるとその旨のメッセージを出力する．

```
module! {
    type: RustFIFO,
    name: "rust_fifo",
    license: "GPL",
}

struct RustFIFO {
    _dev: Pin<Box<Registration<RustFIFO>>>,
}

#[vtable]
impl Operations for RustFIFO {
    fn open(_context: &(), _file: &File)
        -> Result {
        pr_info!("FIFO opened.\n");
        Ok(())
    }
}

Impl Module for RustFIFO {
    fn init(name: &'static CStr,
            _module: &'static ThisModule)
        -> Result<Self> {
        pr_info!("name: {}\n", name);

        Ok(RustFIFO{
            _dev: Registration::new_pinned(
                fmt!("{name}"), ())?,
        })
    }
}
```

図 1　単純なデバイスドライバモジュールの定義

モジュールを構成するソースコードについて，先頭から順を追って述べる．まず，module! マクロを用いて，カーネルモジュールを定義する．type: は，初期化関数を提供するトレイトを定義するための構造体を指定する．また，name: はモジュールの名前，license: はモジュールのライセンスを指定する．これら 3 つのフィールドは最小限必要である．その他必要に応じて，author:, description: なども指定することができる．次に，module! マクロの type: で指定した構造体である RustFIFO を定義している．構造体のメンバとして _dev を定義している．_dev は，登録したデバイスを保持するための変数であり，その型は以下のようになっている．

```
    Pin<Box<Registration<RustFIFO>>>
```

Pin, Box は，どちらも Rust が標準として提供する型である．Pin は，ポインタの参照先が移動しないことを保証するために用いられる．Box は，ヒープにデータ領域を確保するために用いられる．従って，Pin<Box<T>> はヒープに確保された T が移動しないことを表す．Registration は，Rust for Linux の miscdev モジュールにより定義されている型であり，kernel::miscdev::Registration が絶対パスによる表現になる．Rust for Linux が定義する型の絶対パスの先頭には kernel:: が付くことは自明であるため，以下では省略する．miscdev::Registration は，型パラメータとして file::Operations トレイトを実装した型をとるため，Registration<T: file::Operations> と定義されている．ここで T にあたるのが，RustFIFO であるため，RustFIFO は file::Operations

トレイトを実装している必要がある．

RustFIFO での file::Operations トレイトの実装は，以下で定義される．

```
#[vtable]
impl Operations for RustFIFO {
        ...
}
```

file::Operations トレイトは，Linux カーネルにおける file_operations 構造体に対応するインタフェースを定義している．Operations トレイトでの open メソッドの宣言は以下のとおりである．

```
    type Data: ForeignOwnable + Send + Sync = ();
    type OpenData: Sync = ();
    fn open(context: &Self::OpenData, file: &File)
        -> Result<Self::Data>;
```

open メソッドの戻り値の型として Data が，第 1 引数として渡されるデータの型として OpenData が型エイリアスとして宣言されている．それぞれの型のデフォルトは，() と表記される任意の型を表すユニット型になる．また，それぞれの型エイリアス Data, OpenData には，型が実装している必要があるトレイトを指定する，トレイト境界が指定されている．型を+でつなげることで，どちらも必要としていることを表すことができる．即ち，Data は，ForeignOwnable, Send, Sync の 3 つトレイトを実装した何らかの型である必要がある．

RustFIFO の Operations トレイトでは，以下のように，呼び出されるとメッセージを出力するだけの open メソッドのみが実装されている．

```
    fn open(_context: &(), _file: &File)
        -> Result {
        pr_info!("FIFO opened.\n");
        Ok(())
    }
```

RustFIFO の Operations トレイトでの open メソッドの第 1 引数はユニット型，第 2 引数はデバイスファイルへの参照型となっている．戻り値の Ok(()) は，特に値を返さないユニット型の戻り値に，メソッドの戻り値の型 Result に対応させるために Ok() でラップした値となっている．

Operations トレイトの宣言においては，open 以外のメソッドについては，未実装であることを示すエラーを返すだけのデフォルトメソッドの実装が提供されている．open メソッドについては宣言のみで実装が提供されていない．そのため，Operations トレイトの各実装では open メソッドを必ず実装する必要がある．また，Operations トレイトには，属性として #[vtable] が付けられている．これは，各メソッドについて実装の有無を表す定数を生成する．その値から，トレイトの実装がデフォルトメソッドを上書きしているか確認することができ，struct file_operations とのインタフェースにおいて Rust で定義された Operations トレイトのメソッドを呼び出すかどうかの分岐で用いられている．

最後に，Linux カーネルモジュールに必要となるインタフェースを提供する Module トレイトの実装は，以下で定義される．

```
Impl Module for RustFIFO {
    ...
}
```

以下は，Module トレイトの init メソッドの実装である．

```
    fn init(name: &'static CStr,
            _module: &'static ThisModule)
        -> Result<Self> {
    ...
    }
```

init メソッドの第 1 引数はモジュール名の文字列への参照，第 2 引数はモジュールの参照型となっている．&'static は参照の生存期間であり，この場合はモジュールと同じ生存期間となる．

以下は，init メソッドの戻り値である．init メソッドの戻り値の型 Result<Self> に対応して，Self 即ち RustFIFO 型のインスタンスを Ok() でラップした値を返している．

```
    Ok(RustFIFO{
        _dev: Registration::new_pinned(
            fmt!("{name}"), ())?,
    })
```

戻り値の RustFIFO の _dev メンバには，Registration 構造体を new_pinned メソッドによりインスタンス化した値が入る．new_pinned メソッドの第 1 引数はモジュール名，第 2 引数は open メソッドに渡される OpenData 型の値になる．この OpenData 型は，RustFIFO の Operations トレイトでの実装と一致する必要がある．それは，Registration 構造体はトレイト境界として Operations を指定した型パラメータ T を取り，new_pinned メソッドの第 2 引数の型として T::OpenData が指定されているからである．

デバイスドライバモジュールへの FIFO 機能の実装

前節で示したデバイスドライバモジュールに FIFO 機能，即ち書き込んだデータを書き込んだ順番で読み出す機能を実装する．この機能を実装するために，データを保持するための構造体，および RustFIFO の Operations トレイトに read, write 各メソッドの実装を追加した FIFO デバイスドライバモジュールの定義を図 2 に示す．以下，モジュールを構成するソースコードで追加された箇所について，先頭から順を追って述べる．

まず，FIFOdata 構造体は FIFO 機能で必要となるデータを保持するために定義されている．count メンバには保持されるデータのバイト数が入り，buffer メンバには保持されるデータが入る．buffer メンバにより保持されるデータの長さは不定であるため，可変長のバイト列を格納できるようにベクタ型を用いて Vec<u8> 型として宣言されている．

次の FIFO 構造体は，FIFOdata 構造体を Mutex により保護するために定義されている．Operations トレイトで open メソッドの戻り値の型として型エイリアスとして宣言されている Data は，ForeignOwnable, Send, Sync の 3 つトレイトを実装した何らかの型である必要があることを，前節で述べた．その条件を満たすために，Data を FIFO 構造体から Arc<FIFO> とすることに加えて，Mutex により保護することが必要である．

```
module! {
    type: RustFIFO,
    name: "rust_fifo",
    license: "GPL",
}

struct FIFOdata {
    count: usize,
    buffer: Vec<u8>,
}

struct FIFO {
    fifo: Mutex<FIFOdata>,
}

struct RustFIFO {
    _dev: Pin<Box<Registration<RustFIFO>>>,
}

#[vtable]
impl Operations for RustFIFO {
    type Data = Arc<FIFO>;
    type OpenData = Arc<FIFO>;

    fn open(context: &Self::OpenData, _file: &File)
        -> Result<Self::Data> {
        Ok(context.clone())
    }

    fn read(context: ArcBorrow<'_, FIFO>, _file: &File,
            buf: &mut impl IoBufferWriter, _offset: u64)
        -> Result<usize> {
        let mut fifo = context.fifo.lock();
        if fifo.count == 0 {
            return Ok(0)
        }

        let data_len = buf.len().min(fifo.buffer.len());
        buf.write_slice(&fifo.buffer[..data_len])?;
        fifo.buffer.drain(..data_len);
        fifo.count -= data_len;
        return Ok(data_len)
    }

    fn write(context: ArcBorrow<'_, FIFO>, _file: &File,
             buf: &mut impl IoBufferReader, _offset: u64)
        -> Result<usize> {
        let data = buf.read_all()?;
        let data_len = data.len();
        let mut fifo = context.fifo.lock();

        fifo.buffer.try_extend_from_slice(&data[..])?;
        fifo.count = fifo.buffer.len();
        Ok(data_len)
    }
}

impl Module for RustFIFO {
    fn init(name: &'static CStr,
            _module: &'static ThisModule)
        -> Result<Self> {
        let mut fifo = Pin::from(UniqueArc::try_new(FIFO {
            fifo: unsafe {
                Mutex::new(FIFOdata {
                count: 0,
                buffer: Vec::new(),
            })},
        })?);

        let pinned = unsafe {
            fifo.as_mut().map_unchecked_mut(|s| &mut s.fifo)
        };
        kernel::mutex_init!(pinned, "FIFO::data");

        Ok(RustFIFO{
            _dev: Registration:: new_pinned(
                fmt!("{name}"), fifo.into())?,
        })
    }
}
```

図 2　FIFO デバイスドライバモジュールの定義

RustFIFO 構造体は図 1 と同じである．その詳細は前節で述べたとおりである．

Operations トレイトの実装では，open メソッドの戻り値の型エイリアスである Data，および第 1 引数として渡されるデータの型エイリアスである OpenData が，Arc<FIFO> と定義されている．そのため，open メソッドの第 1 引数が OpenData 型となっており，戻り値が Data 型となっている．open メソッドの第 1 引数は，Module トレイトの init メソッドにおいて Registration 構造体を new_pinned メソッドによりインスタンス化する際の第 2 引数の値が渡される．つまり，Module トレイトの init メソッドにおいて生成，初期化された Arc<FIFO> 型のインスタンスが open メソッドに渡されることになる．

Operations トレイトには，read および write メソッドも実装されている．read および write メソッドの第 1 引数には open メソッドの戻り値が渡されるため，Data 型をもとにした型が構成されている．そのため，FIFO 構造体が渡されることになる．FIFO 構造体の実際のデータは Mutex により保護された FIFOdata 構造体の中にある．そこで，Mutex をロックすることで，FIFOdata 構造体にアクセスすることができる．以下では，FIFO 構造体の fifo メンバが保持する Mutex をロックし，FIFOdata 構造体を取り出している．

```
    let mut fifo = context.fifo.lock();
```

取り出した FIFOdata 構造体は変更可能であるため，変数は mut と宣言されている．

Operations トレイトの read メソッドの第 3 引数は，読み出すデータの書き込み先として IoBufferWriter 型のバッファが指定されている．read メソッドによる FIFO からの読み出しは，FIFOdata 構造体の buffer メンバに保持されているデータの IoBufferWriter への書き込みになる．IoBufferWriter にデータ書き込む際のデータの長さは，IoBufferWriter のバッファの長さが最大となる．そのため，FIFOdata 構造体の buffer メンバに保持されているデータの長さと比較し，どちらか短い方を書き込むデータの長さとしている．IoBufferWriter に書き込んだデータは FIFOdata 構造体の buffer メンバからは削除する．そのため，drain メソッドに書き込むデータの長さを指定し，buffer メンバの先頭からデータを削除している．

Operations トレイトの write メソッドの第 3 引数は，書き込まれるデータの読み出し先として IoBufferReader 型のバッファが指定されている．write メソッドによる FIFO への書き込みから

は，読み出しとは逆に，IoBufferReader のバッファから読み出したデータの FIFOdata 構造体の buffer メンバへの書き込みになる．buffer メンバは可変長のバイト列を格納できる Vec<u8> 型であるため，IoBufferReader から読み出したデータを全て buffer メンバへの書き込むことができる．

Module トレイトの init メソッドでは，FIFO 構造体のインスタンスを作成している．FIFO 構造体は，Mutex により保護された FIFOdata 構造体の中に実際のデータを保持している．Mutex は Pin 型でくるむことでデータ領域が移動しないことを保証する必要がある．また，Mutex は初期化が必要である．そのため，Pin 型でくるまれた FIFO 構造体の中から Mutex を取り出したうえで，初期化のために mutex_init! マクロを呼び出している．

5 考察

本章では，4 章で述べた Rust 言語による Linux デバイスドライバの開発について考察する．

Mutex は初期化しないと実行時のエラーが発生した．Rust はメモリ安全性を提供すると言われている．メモリ安全性とは，プログラムの不正なメモリ操作による問題が起きないようにする安全性である．しかしながら，初期化されていない Mutex が存在しても，問題なくコンパイルができ，特に警告も出力されなかった．そして，実行時にエラーが発生してしまった．この Mutex は Rust for Linux で開発されたものである．このような問題が発生するのは，Rust for Linux がまだ開発途上であるためかもしれないが，Rust でもこのような問題が発生しうることがわかったことは，1 つの有用な知見となった．

Registration 構造体をインスタンス化する際の第 2 引数が，Operations トレイトの open メソッドの第 1 引数として渡されることで，これらの引数の型が合致している必要がある．しかしながら，デバイスの登録を行う Module トレイトの init メソッドと Operations トレイトの open メソッドという，役割も異なるトレイトの間でこのような関係が生じることは，決して分かりやすいとは言い難い．これらの関係は 4.1 節で述べたように，miscdev::Registration が，型パラメータとして file::Operations トレイトを実装した型を取るように，Registration<T: file::Operations> と定義されていることから生じている．このような型の関係性について，強い片づけを行う言語ではどのように取り扱っているのか調査する必要がある．

6 おわりに

本論文では，安全性の高いシステムプログラミング言語である Rust を用いて，Linux カーネルのデバイスドライバを開発する方法について述べた．Rust for Linux が提供する Rust サポートの機能を用い，FIFO 機能を実装したデバイスドライバの開発を行うことで，実際に動作するデバイスドライバの開発をおこなった．Rust によるデバイスドライバの開発方法についての情報はまだ限定的であるため，モジュールを構成するソースコードの各行を詳細に調べることで，Rust for Linux による Rust サポートについて多くの知見を得ることができた．今後は，デバイスドライバの本来の役割であるハードウェアデバイスの制御を行うデバイスドライバを Rust で開発し，さらに知見を深めていく予定である．

参考文献

1. Palix N, Thomas G, Saha S, Calvés C, Muller G, Lawall J. Faults in Linux 2.6. ACM Trans Comput Syst. 2014;32. doi:10.1145/2619090

2. Rust for Linux. Available: https://github.com/Rust-for-Linux/

3. Linux-next. Available: https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=c7c4c9b88eecb28d3b48ba855cd6f9f7391a33b2

4. Ojeda M. [RFC] Rust support. 2021. Available: https://lkml.org/lkml/2021/4/14/1023

5. 追川 修一. プログラミング言語 Rust によるカーネルモジュールの開発. 東京都立産業技術大学院大学紀要. 2022;15: 173–180.