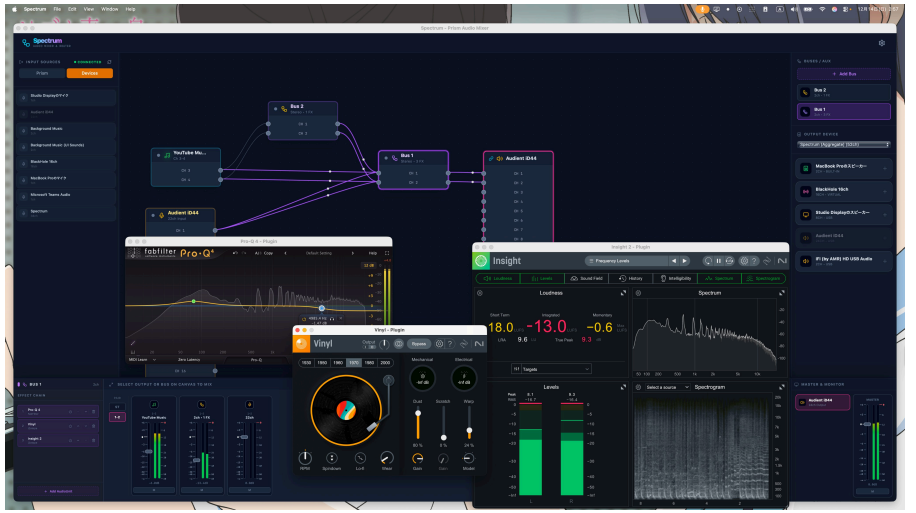
# Rust製OSにLKM的な機能を実装したら
# いい勉強になった話

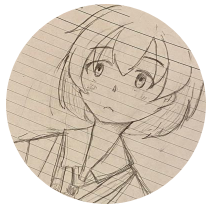
**@petitstrawberry**

2026/05/09 Kyoto.rs #1

# 自己紹介

- 立命館大学 M2 情報理工学研究科, M研究室
  - OSとかハイパーバイザの研究
- ヤフオクで1.5万円で落としたSiFive HiFive Premier P550 →
- 趣味
  - 情報系全般
    - 自作OS, オーディオルータ&ミキサー, 色々なツール, etc...
    - 自宅鯖, ヤフオク漁り, おうちクラウド(k8s), etc...
  - 音楽
    - フルート, ベース, シンセ

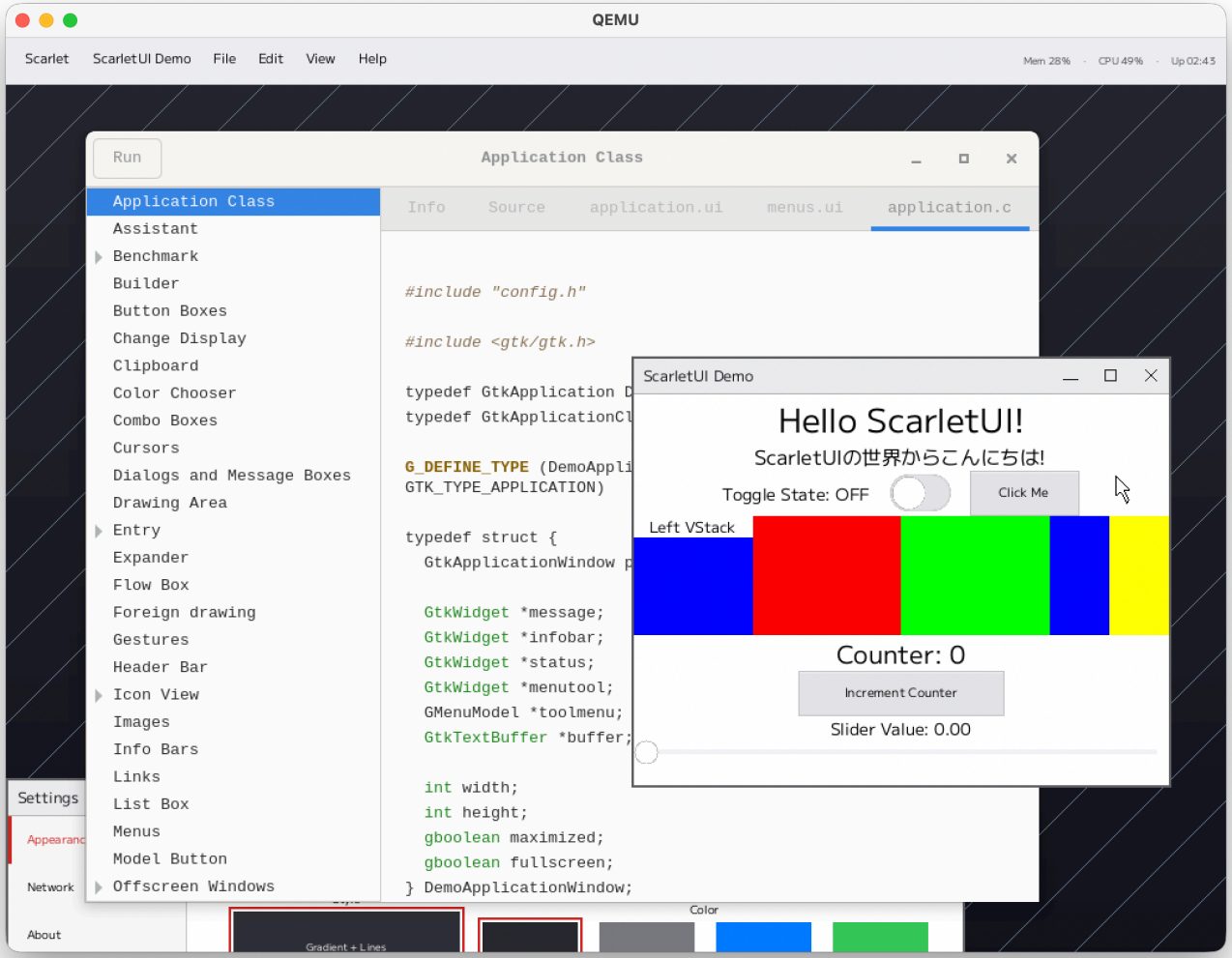


X: @petitstb
GitHub: petitstrawberry

# 自作OS Scarlet[1]

A kernel in Rust designed to provide a universal, multi-ABI container runtime.と銘打ってますが
その実態は思いついたことを詰め込んだキメラ

- **Rust製**
  - そこそこ大規模な方かも
    - 伏線
- **複数OS向けバイナリの透過的な実行**
  - ユーザやプログラムは何も意識せずにexecやshellを使うだけで,
    色々なOS向けのバイナリが動く
    - 異なるOSバイナリ間のIPCなども可能
  - 右の図はScarlet上でScarletアプリとLinuxアプリが同時に動いている様子




[1] Scarlet: https://github.com/petitstrawberry/Scarlet

# カーネル巨大化問題

個人の趣味で作っているとはいえ, 機能的にもコード量的にもどんどん巨大になっていく

- バイナリ互換機能 (ABI Module)
  - Scarlet, xv6, Linux (WIP), Windows (pending), Darwin (macOS) (WIP)
- ドライバ
  - 割込みコントローラ (PLIC, GICv3)
  - タイマ (CLINT, ARM Generic Timer)
  - VirtIO (block, net, gpu, etc...)
  - USB (xHCI), PCI
  - Apple Silicon向けの周辺機器 (PMU, GPIO, etc...)
- ファイルシステム, ネットワークスタック, etc...
- ハイパーバイザ機能

- size的な話
  - ソースコード: 10万行以上
- ビルド時間
  - 諸々含めると数分かかってくる

ドライバの変更や, ちょっとした機能追加のたびに毎回数分待つのは辛いし,
使わないドライバやカーネル機能がずっとメモリに常駐しているのもあまりよろしくない

# そこでLKM的な機能を実装してみたくなる

## LKM (Loadable Kernel Module)とは

Linuxなどで使われるカーネルの仕組みで, ドライバなどの機能を動的にカーネルにロードできるようにするもの

```
$ modprobe my_driver.ko
```

こんな感じに**実行時に**カーネル空間にドライバなどをロードできるようになる

こうすることで,

- カーネル自体にドライバを全てstaticリンクする必要はなくなり, サイズの削減やビルド時間の短縮が期待できる
- initで必要なドライバだけロードするようにすれば, メモリの節約もできる
- ドライバ開発も, 都度カーネルをビルドするのではなく, モジュールだけビルドしてロードすれば良くなるので, 開発効率も上がる

# LKMの仕組み (簡単に解説) （1/2）

まずはLKMの仕組みを知らないと真似できない

**カーネルモジュール作成フロー**

1. ドライバなどの機能を実装して, カーネルモジュールのソースコード(.c)を作る
   - ここで, カーネルが公開する関数やデータ構造が利用可能
     - EXPORT_SYMBOLなどでカーネルが公開しているシンボルを利用できる
2. モジュールをコンパイルしてET_RELな(再配置可能な) **オブジェクトファイル(.o)** を作る
   - この段階では, まだ全てのシンボルが解決されているわけではない
   - 特に, カーネルが公開しているシンボルは, カーネルモジュールのビルド時には存在しないので, これらは未解決のままになる
3. 諸々のメタデータ的なものを追加する作業を経て, **カーネルモジュール(.ko)** が完成

# LKMの仕組み (簡単に解説) （2/2）

## カーネルモジュールのロードフロー

1. カーネルはモジュール(.ko)のELFを読み込む
2. 適当にメモリ上に配置
3. シンボルを解決&再配置
    - これは, ko内部のシンボルだけでなく, koが依存しているカーネルのシンボルも解決する必要がある
    - この時, カーネルは自身のシンボルテーブルを参照して, 未解決のシンボルを解決する
4. モジュールのinit関数を呼び出す

じゃあ、これをRust製OSであるScarletに実装してみよう、というのが今回の話

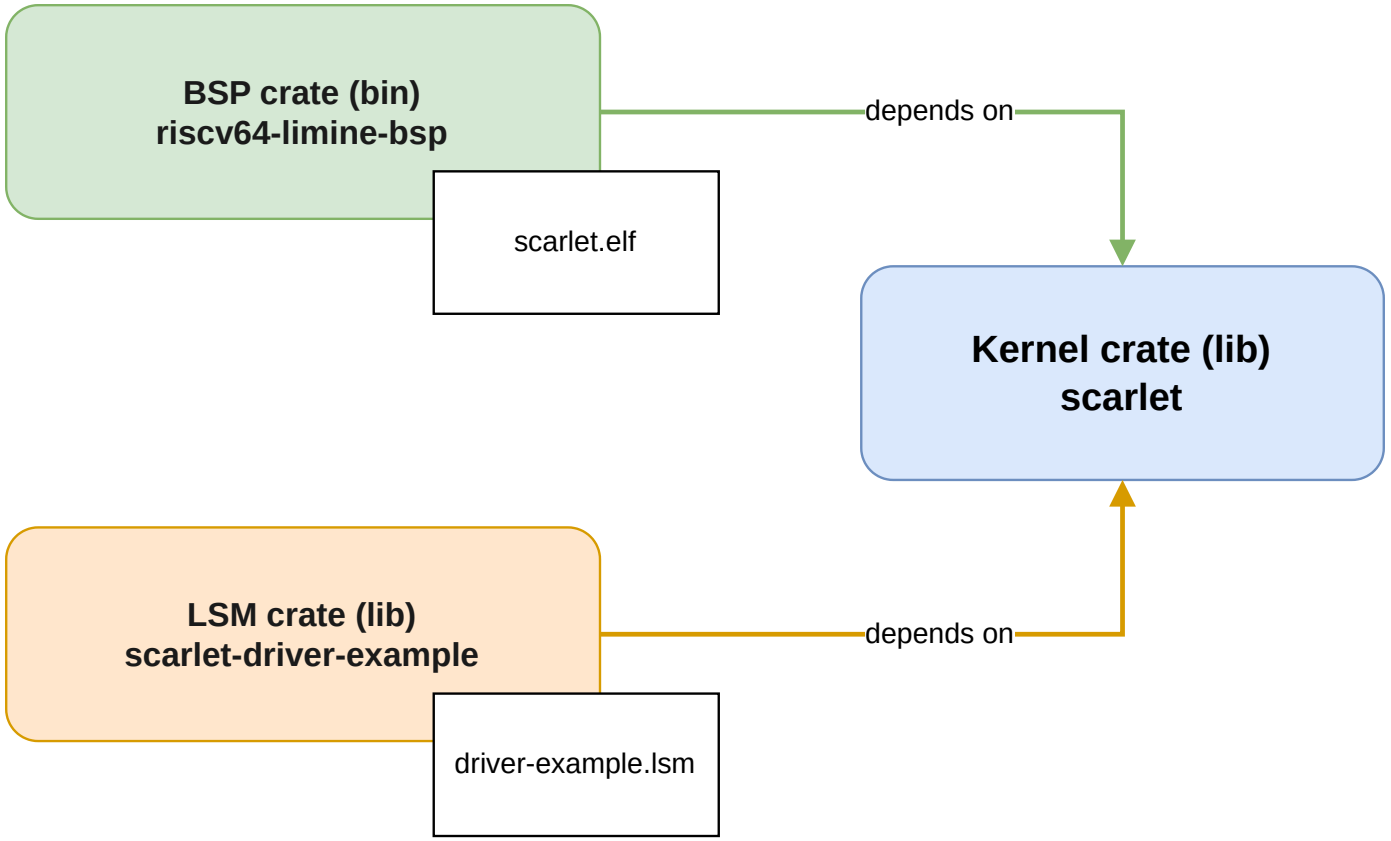
Rustで真似するだけよね, 余裕余裕

そもそも, このロード機構自体を実装するのが辛くないですか？という話もあるが, まあそれは置いておいて今回は言語的な問題に絞って話す

# 前提

Scarletの依存関係はこんな感じ

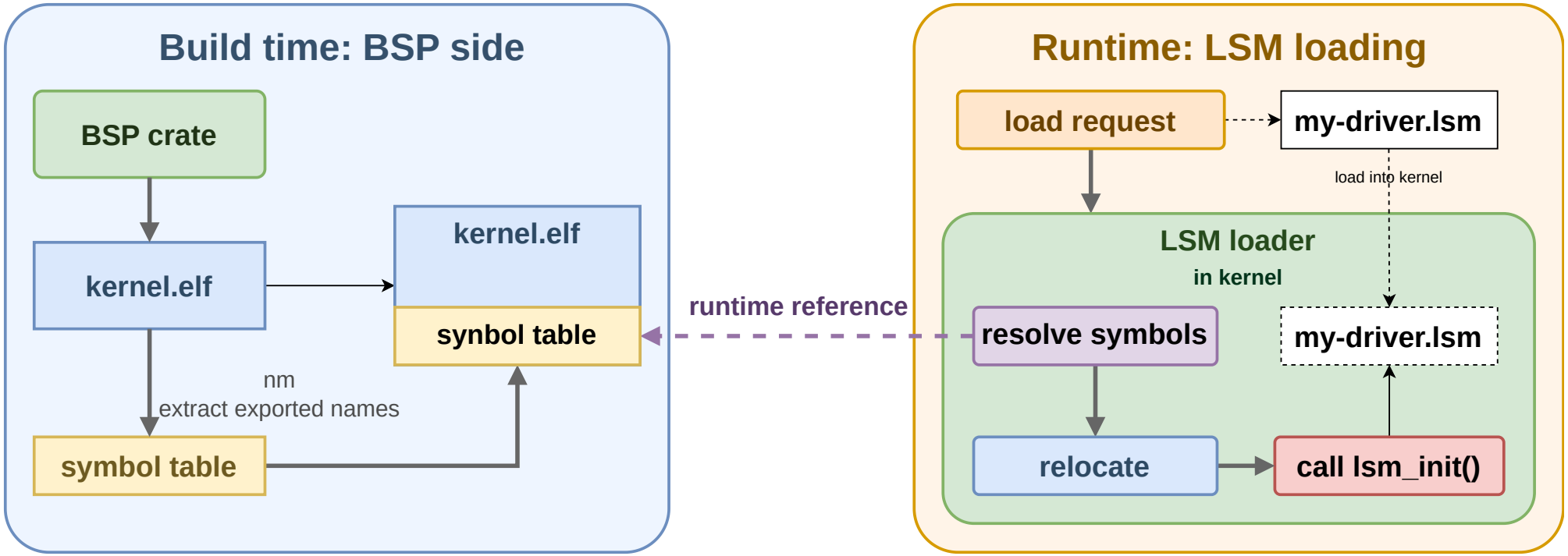
- BSP (Board Support Package)
  - ブートなど, ボード固有コード
  - これが実際の実行バイナリを生成
- LSM (Loadable Scarlet Module)
  - LKM的な機能を提供するモジュール
  - こいつが今回のメイン
- Kernel
  - OSのコア機能を提供
  - ほぼ全部ここ
  - こいつはライブラリとして提供される
  - BSPやLSMはこれを利用して実装される




**Scarlet crate dependencies**

# Scarletで真似する

1. まず, シンボル名を公開する機構を実装
    1. BSPクレートのビルド完了後, ELFからnmコマンドでシンボルテーブルを抜き取る
    2. ELFの適切なセクションにシンボルテーブルを配置して, 実行時に参照可能にする
2. カーネル側にLSMをロードする機構を実装
    - 仕込んでおいたシンボルテーブルを参照して, LSMの未解決シンボルを解決する



# ダメそう

とはいえ, そうは簡単に行かなかった！！

↓ScarletのShellからLSMをロードしてみる

```
# lsm_load lsm-test
loading module: /scarlet/system/scarlet/modules/lsm-test.lsm
[LSM] unresolved symbol: _RNvNtNtNtCs5hMFX2pKYKj_7scarlet7library3std5print6__print
failed to load lsm-test: unresolved symbol (12)
#
```

**シンボルが見つかりません！！？？**

実際のカーネルのシンボルとLSMの未解決シンボルを見比べてみると, どうも名前が違う

- LSM側の未解決シンボル: `_RNvNtNtNtCs5hMFX2pKYKj_7scarlet7library3std5print6__print`
- カーネル側が公開しているシンボル:
  `_RNvNtNtNtCs33NknshCXnf_7scarlet7library3std5print6__print`

なんか良くわからんhashみたいなのが前についてるんですが

# Name Mangling (1/2)

Rustのコンパイラが関数や変数のシンボル名を特定の規則に従って変換すること

今回の例で言うと

```
_RNvNtNtNtCs5hMFX2pKYKj_7scarlet7library3std5print6__print
          |`----+-----`||`--+--`
          |     |      ||   |
          |     |      ||   +------------------ crate-root identifier "scarlet"
          |     |      |+---------------------- length 7 of "scarlet"
          |     |      +----------------------- end of base-62-number
          |     +-------------------------------disambiguator for crate-root "scarlet" Cs5hMFX2pKYKj
          +------------------------------------ crate-root
```

- このmanglingには規則性があってv0 Symbol Format[2]に従っている
  - RFC2603で定義されているので詳細はそちらを参照
  - 昔はC++互換の別の規格だったが今はv0が主流
    - ただし, Rustとしてはこれで安定してるわけではないらしい

[2] v0 Symbol Format: The rustc-book, https://doc.rust-lang.org/rustc/symbol-mangling/v0.html

# Name Mangling (2/2)

```
_RNvNtNtNtCs5hMFX2pKYKj_7scarlet7library3std5print6__print
          |`----+-----`||`--+--`
          |     |      ||   |
          |     |      ||   +------------------ crate-root identifier "scarlet"
          |     |      |+---------------------- length 7 of "scarlet"
          |     |      +----------------------- end of base-62-number
          |     +-------------------------------disambiguator for crate-root "scarlet" Cs5hMFX2pKYKj
          +------------------------------------ crate-root
```

- crate disambiguator
  - 同じクレート名でも区別可能にする
  - クレートの内容に基づいて生成されるハッシュ値で, クレートの内容が変わると変わる
  - ソースコードどころか, **featureフラグやコンパイルオプションを変えるだけでも変わる**
- BSP側 (カーネルバイナリを生成する側): debug buildのopt-level=3
- LSM側 (カーネルモジュール側): release buildのopt-level=0

なので, rustc的には全く別のクレートと見なされている

# Rust ABIは不安定

**C言語マンの発想**「最適化されてても, 構造体のフィールドや関数のシグネチャが同じなら実際同じなんだから, manglingとか無視したらいいじゃん」

**Rust**「あかん。」

そもそもRustは...**安定したABIが存在しない**

- Name Mangling (シンボル名の装飾)
- Data Layout (構造体のレイアウト)
- Calling Convention (関数呼び出し規約)

これらは, 同じソースコードでもrustcのバージョンによってもコロコロ変わる。最適化オプションとかでも変わるかも

Manglingであれだけ厳密にクレートを区別してるのも, そういう理由があるから。多分

---

実際のところ最適化の自由度を増やすためとかそういう理由らしいが, LLVMバックエンドなのでどこまでrustc側でコントロールしてるかはしらん...

# じゃあ、さ...

**「crate-typeとしてdylibが指定可能なのは何のためにあるんだよ...結局, 独立してビルドした環境では使い物にならんはずでしょ....」**

→ Rustコンパイラ(rustc)と動的リンクするような用途を想定してる

- 身近な例で言うとproc macroクレート
  - これもdylibとしてビルドされる
  - rustcがこのdylibをロードして, 手続型マクロのコードを実行する

**「じゃあ、普通にRustで動的にロードする共有ライブラリ作りたい場合はどうしたら？」**

→ crate-typeをcdylibにして, C言語のABIでシンボルを公開する
ただし, C ABIに縛られる

- `struct␣#[repr(C)]` したものだけ
- 関数も `extern "C"` + `#[no_mangle]` で定義する必要がある
- これらはC言語で表現可能なものに限られるので, Rustの表現力は失われる

# 諦め

- `extern "C"` + `#[no_mangle]` や `#[repr(C)]` をつけて, cdylibにするのは嫌
- RustはRustのまま動的にロードしたい

→ もう, **rustcというかtoolchainバージョン, featureなど, rustflagsを合わせるしかない**

そもそも, 今回はLKM的なものを作りたいという話であった

- LinuxのLKMももちろんカーネルツリーは同じである必要がある
  - これに, カーネルのfeatureフラグや最適化オプションも固定にするくらいの制限が増えてもまあいいかとする
  - Linuxですら厳しい制限なんだから, 今更いいでしょうという感じで妥協
- 元々の動機はビルド時間の短縮やメモリ使用量の削減だったので, これで目的は達成可能

こういうのは独自のビルドツールcargo-scarletっていうので吸収することにした

結局Linuxも, カーネルバージョンへの追従のためにDKMS（Dynamic Kernel Module Support）に頼っているので, まあいいでしょう...

# 一応, 結果

LSM側のコード例 (細かいメタデータ類とかは省略)

```
#![no_std]

use scarlet::println;

// □ントリポイントだけはno_mangleで定義□□ (□□□□□呼び出□□□)
#[unsafe(no_mangle)]
pub extern "C" fn scarlet_lsm_init() -> Result<(), &'static str> {
    println!("[lsm-test] Loadable Scarlet Module loaded successfully!");
    Ok(())
}
```

これで, LSMをビルドして, ScarletのShellからロードしてみると

```
# lsm_load lsm-test
loading module: /scarlet/system/scarlet/modules/lsm-test.lsm
[lsm-test] Loadable Scarlet Module loaded successfully!
module loaded successfully
#
```

# おわりに

LKM的な機能の実装は, RustのABIの不安定さやName Manglingの問題など, 色々な問題があって大変だった

次回, 「Rust製OSでDKMS的なものを実装する」
(自作OS上でrustcを動かして, セルフホストすることになります)

できる気がしません (やるとは言ってない)

ScarletのGitHubリポジトリ
https://github.com/petitstrawberry/Scarlet