DOL-12

Rust で自作して学ぶ コンテナの仕組み

苅野 秀和

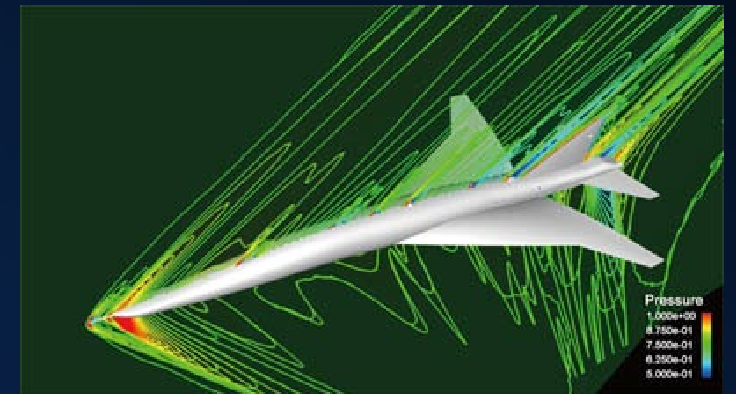
アマゾンウェブサービスジャパン
ソリューションアーキテクト

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

苅野 秀和 (かりの ひでかず)

https://bsky.app/profile/hkford.bsky.social

- ウェブ系企業担当で Rust とコンテナが好きなソリューションアーキテクト
- 昔は飛行機の研究をしていた
- 時々 Zenn に記事を投稿しています
  https://zenn.dev/hkdord



研究対象だった飛行機

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

本セッションのお品書き

- コンテナの仕組み
- Rust で自作
- デモ
- まとめ

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

コンテナの仕組み

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

コンテナ=リソースが隔離されたプロセス

- コンテナはリソースが隔離された環境で実行されるプロセス
- 隔離されていないと他のコンテナのリソースが見えてしまったり一つのコンテナがリソースを消費しすぎる恐れ
(PC 作業をしていてブラウザとエディタとチャットアプリを動かしていると動作が重くなるイメージ)
- Docker の場合コンテナによる CPU やメモリの使用量を制限できる(ref: https://docs.docker.com/config/containers/resource_constraints/)
- Amazon ECS の場合 task size で制限できる(ref: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html#task_size)

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

リソースの隔離方法

> Docker container technology was launched in 2013 as an open source Docker Engine.
>
> It leveraged existing computing concepts around containers and specifically in the Linux world, primitives known as cgroups and namespaces. Docker's technology is unique because it focuses on the requirements of developers and systems operators to separate application dependencies from infrastructure.

https://www.docker.com/resources/what-container/

Docker の場合 Linux kernel の機能(cgroups や namespaces など)を活用して隔離を実現している。

本セッションでは Rust でコンテナを実装しコンテナの裏側をチラ見します。

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

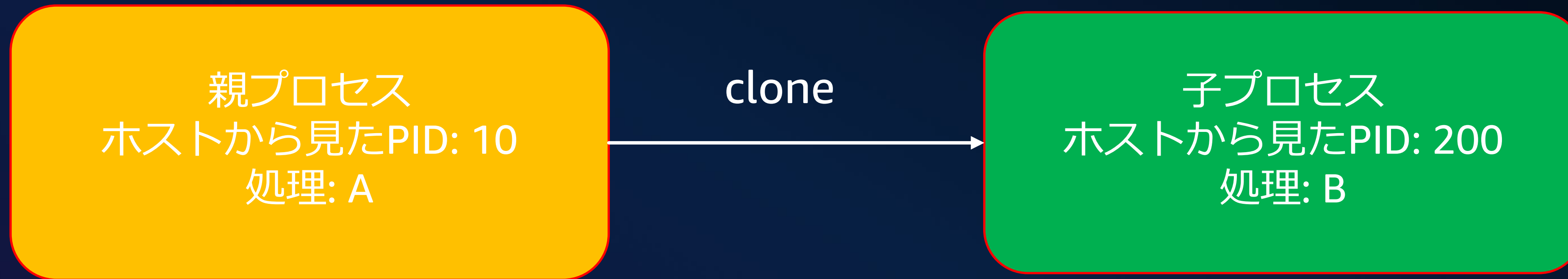
コンテナの仕組み

- コンテナはリソースが隔離された環境で実行されるプロセス(2 回目)
- プロセスなので clone システムコール で生成できる
- その上でリソースを隔離する
  - root のマウントポイントを隔離 -> pivot_root システムコール
  - 他のプロセスのリソースを見られないようにする -> namespace の利用
  - リソースの使用量に制限をかける -> cgroups の利用
  - 権限を細かく制御する -> capabilities の利用

今回は clone, namespace, cgroups を見ていく

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

clone



clone によって新しいプロセスを生成する

fork と異なり親子プロセスの間で共有される実行コンテキストを細かく制御できる(子プロセスを新しい namespace で起動するかどうかなど)
clone の引数で子プロセスの中で行う処理(図の処理 B)を指定可能
ref: https://man7.org/linux/man-pages/man2/clone.2.html

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

namespace

PID namespace M

親プロセス
ホストから見たPID: 10

PID namespace N

子プロセス
子プロセスから見たPID: 10
(ホストから見たPID: 200)

namespace によってリソースをお互いに見えなくすることができる。

例えば PID namespace だとプロセス ID (PID) を namespace ごとに持つことができる。

namespace が異なれば同じ PID を持ってもいいため、上の図にあるように同じ PID: 10 を持つことができている。

あくまで子プロセスから PID:10 に見えるだけであって、ホストから見ると PID は重複していない(ホストから見た子プロセス PID は 200)

aws

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

cgroups

/sys/fs/cgroup/td/cgroup.procs

子プロセス
CPU を 10% 割り当て

CPU、 メモリ、ネットワークなどのリソースを使用するプロセスをグループ化し
リソース使用量を制限するもの。

cgroupfs という特殊なファイルシステムを通じて利用する。
プロセスディレクトリ /proc に存在するファイルを参照してプロセスの詳細を
調べられるようなイメージ

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Rust で自作

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Rust

はじめる

バージョン 1.78.0

効率的で信頼できるソフトウェアを
誰もがつくれる言語

なぜRustか？

パフォーマンス

Rustは非常に高速でメモリ効率が高くランタイムやガベージコレクタがないため、パフォーマンス重視のサービスを実装できますし、組込み機器上で実行したり他の言語との調和も簡単にできます。

信頼性

Rustの豊かな型システムと所有権モデルによりメモリ安全性とスレッド安全性が保証されます。さらに様々な種類のバグをコンパイル時に排除することが可能です。

生産性

Rustには優れたドキュメント、有用なエラーメッセージを備えた使いやすいコンパイラ、および統合されたパッケージマネージャとビルドツール、多数のエディタに対応するスマートな自動補完と型検査機能、自動フォーマッタといった一流のツール群が数多く揃っています。

https://www.rust-lang.org/ja/ より

aws

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Rust の特徴 – unsafe

```rust
46     /// Get system identification
47     pub fn uname() -> Result<UtsName> {
48         unsafe {
49             let mut ret = mem::MaybeUninit::zeroed();
50             Errno::result(libc::uname(ret.as_mut_ptr()))?;
51             Ok(UtsName(ret.assume_init()))
52         }
53     }
```

https://github.com/nix-rust/nix/blob/master/src/sys/utsname.rs#L46

unsafe Rust でかくとメモリ安全性をプログラマが保証することで Rust が推奨していない機能を使うことができる。 他の言語(C など)を Rust から呼び出すなど。
上記の例だと unsafe ブロックの中で libc::uname を Rust から呼び出している。uname 関数の呼び出しは safe で Result<UtsName> が返ってくる

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

今回お世話になったクレート

- nix(https://github.com/nix-rust/nix): libc が公開する unsafe な API の代替で safe な API を提供

nix クレートを使えばシステムコールを実行することはできるが、いくつかの操作はさらに抽象化した以下のクレートを使用

- capctl(https://github.com/cptpcrd/capctl): capabilities を操作
- syscallz(https://github.com/kpcyrd/syscallz-rs): seccomp を操作
- cgroups-rs(https://github.com/kata-containers/cgroups-rs): cgroup を操作

aws

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

nix クレートの特長

```
// libc api (unsafe, requires handling return code/errno)
pub unsafe extern fn gethostname(name: *mut c_char, len: size_t) -> c_int;

// nix api (returns a nix::Result<OsString>)
pub fn gethostname() -> Result<OsString>;
```

https://github.com/nix-rust/nix

C のコードを呼び出す libc クレートは unsafe でプログラマ側でメモリ安全性を保証したり errno からエラーの種類を判定する必要がある。
nix クレートは libc の機能をラップして型をつけて可能な限り safe に扱えるようにしている。
今回はシステムコールを実行することが多く safe に扱えるので嬉しい

aws

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

clone の場合(nix クレートを利用)

```rust
pub unsafe fn clone(
    cb: CloneCb<'_>,
    stack: &mut [u8],
    flags: CloneFlags,
    signal: Option<c_int>
) -> Result<Pid>
```

```rust
pub type CloneCb<'a> = Box<dyn FnMut() -> isize + 'a>;
```

Available on **crate feature sched** only.

[−] Type for the function executed by `clone`.

https://github.com/nix-rust/nix

clone システムコールでプロセスを生成できる。nix::sched::clone は clone をラップしていて引数の cb に生成したプロセスの中で実行される関数を指定できる
今回のデモだとコンテナを作る = この clone でプロセスを作ることと言っても過言ではない

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

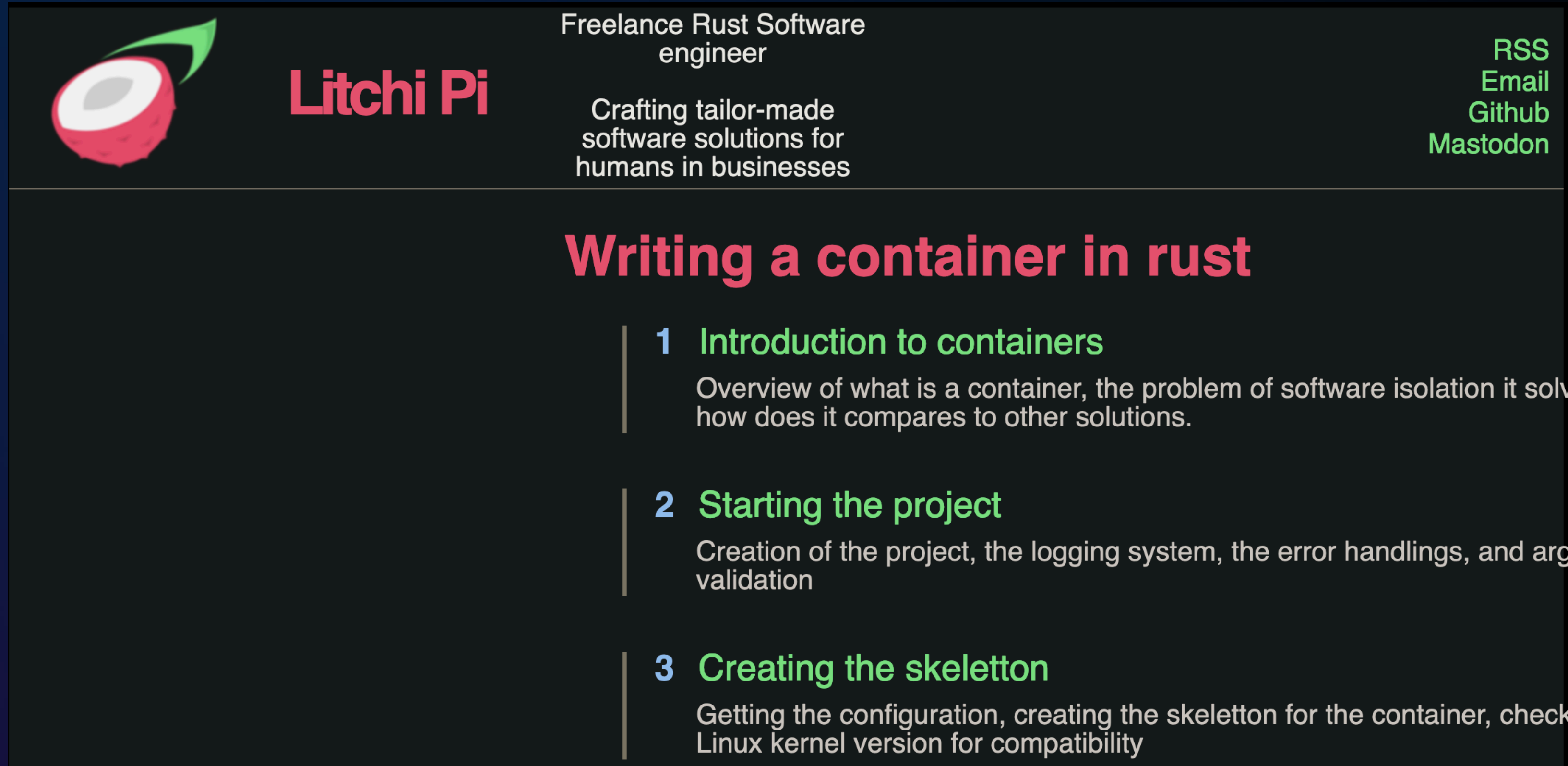
cgroup の場合(cgroups-rs を利用)

```
let cgs = CgroupBuilder::new(hostname)
    // quota method writes to cpu.max: https://docs.rs/cgroups-rs/0.3.4/src/
    // It's in the following format: $MAX $PERIOD which indicates that the g
    .cpu()
    .period(CPU_PERIOD)
    .quota(MAX_CPU_BANDWIDTH)
    .done()
    // Limiting the memory usage to 1 GiB
    // The user can limit it to less than this, never increase above 1Gib
    .memory()
    .kernel_memory_limit(KMEM_LIMIT)
    .memory_hard_limit(MEM_LIMIT)
    .done()
    // This process can only create a maximum of 64 child processes
    .pid()
    .maximum_number_of_processes(MAX_PID)
    .done()
    // Give an access priority to block IO lower than the system
    .blkio()
    .weight(50)
    .done()
    .build(Box::new(V2::new()));
```

Builder pattern で操作できる
CPU やメモリなどのリソースに制限をかけることが可能
例えば図の例だと cpu().quota() で CPU の period (スケジュール期間) のうち使える時間を指定できる

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

参考にしたブログ



https://litchipi.github.io/series/container_in_rust
を一部改変しています

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

デモ
~cgroup で CPU 使用率を 10% に制限する~

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

今回お見せするデモ

clone システムコールで生成した子プロセス(これがコンテナのつもり)
に cgroup で CPU 使用率を 10% に制限する
その上で子プロセスに負荷をかけて、ホストから見た
プロセスの負荷を観察する

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

まとめ

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

セッションまとめ

- コンテナはホストマシンから見ると 1 つのプロセス
  (これ言うの何回目だろう)
- コンテナは Linux kernel の機能(cgroup, namespace など)を使って
  リソースを隔離している
- 自作すると理解が深まる
- Rust はいいぞ

aws
© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Thank you!

苅野 秀和

hidekari@amazon.co.jp

© 2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.