2024 年度

筑波大学情報学群情報科学類

情報特別演習 I 最終報告書

題目

OCI Runtime Specification に準拠した非特権コンテナランタイムの開発

所属 情報科学類 2 年

著者 川崎 晃太朗

アドバイザ教員 新城 靖

要　旨

本研究では、OCI Runtime Specification に準拠した非特権コンテナランタイム「tiny-youki」の開発を行った。近年、コンテナ技術はソフトウェアの開発および運用において不可欠な要素となっており、その中核を担うコンテナランタイムの理解と実装に関心が集まっている。しかし、既存のコンテナランタイムは複雑で大規模なコードベースを持つものが多く、特に非特権環境で動作するコンテナランタイムの開発に関する詳細な情報は十分に共有されていない。本研究では、この課題を解決するために、非特権環境で動作可能なコンテナランタイムをシンプルな実装で開発し、理解しやすい形で実装方法を示すことを目的とした。

開発に際しては、既存の Rust 言語によるコンテナランタイム「youki」を参考にしながら、非特権環境に適した設計を採用した。特に、User 名前空間の活用により、非特権にコンテナを起動可能とし、さらに Mount, PID, UTS などの各種名前空間の分離を実装することで、非特権環境下におけるコンテナの隔離性を確保した。また、OCI Runtime Specification に準拠することで、標準仕様に沿ったコンテナ実行を可能にし、既存のコンテナエコシステムとの互換性を維持した。

開発した「tiny-youki」は、非特権環境での動作を前提としながらも、コードのシンプルさを重視し、コンテナランタイムの基本的な動作原理を容易に理解できる構成とした。本研究の成果は、非特権コンテナランタイムの設計・実装に関する知見を提供し、今後のコンテナ技術の発展および教育用途に寄与するものである。

目次

第 1 章　はじめに

1.1 背景・動機

近年、コンテナ技術はソフトウェアの開発および運用において不可欠な要素となっており、その基盤を支えるコンテナランタイムの仕組みに関心を持った。特に、「100 行でコンテナランタイムを実装する」という内容の Web 上に公開されたチュートリアル[1] に取り組んだことを契機として、より詳細な理解を深めたいと考えるようになった。

加えて、本学の情報科学類におけるコンピューティング環境で動作可能なコンテナランタイムを開発することに興味を抱いた。しかし、学内環境でコンテナを実行するためには、管理者権限を必要としない「非特権」で動作するコンテナランタイムを実装する必要がある。現状では非特権コンテナランタイムを自作するための詳細なチュートリアルは少なく、その実現方法についての知見は十分に共有されていない。このような状況を踏まえ、筆者自身が非特権コンテナランタイムを開発することで、学習を進めるとともに、同様の関心を持つ他の学生や研究者にとっても有益な情報を提供できるのではないかと考えた。

1.2 目的

本研究では、既存の非特権コンテナランタイムの実装を参考にしながら、シンプルかつ理解しやすいコードで非特権環境に対応したコンテナランタイムを自作することを目標とする。これは、いわゆる「車輪の再発明」である。

特に、以下の点を重点的な目標として設定する。

1. **シンプルで理解しやすい実装**

コンテナランタイムの基本的な動作原理を明確にし、過度な複雑性を排除することで、学習および解析が容易なコードを開発する。

2. **非特権環境での動作**

コンピューティング環境において非特権に実行可能なコンテナランタイムを実装し、非特権環境でのコンテナ実行の実現可能性を検証する。

3. **OCI Runtime Specification への準拠**

開発するコンテナランタイムを OCI Runtime Specification に準拠させることで、標準仕様に沿った設計・実装を行い、保守性と拡張性を高める。

[1] https://www.infoq.com/articles/build-a-container-golang/

本研究を通じて、非特権コンテナランタイムの仕組みについての理解を深めるとともに、その実装に関する知見を蓄積し、今後のコンテナ技術の発展に寄与することを目指す。

第 2 章　演習内容

2.1 概要

本演習では、コンテナランタイム「tiny-youki」を開発した。tiny-youki は、既存のコンテナランタイム「youki」[1] を参考にした非特権コンテナランタイムである。youki は、OCI Runtime Specification [2] に準拠し、非特権環境で動作可能な Rust 言語によるコンテナランタイムである。Rust 言語は、安全性とパフォーマンスを重視したシステムプログラミング言語であり、システムコールやプロセス管理のための低レベル API を提供している。この特性により、コンテナランタイムの開発に適している。

youki のコードは約 4 万行もあり、コードリーディングには時間がかかる。一方 tiny-youki では、youki のコードベースを活用しつつ、より小規模でシンプルなコードベースを目指して開発した。このため、「tiny-youki」という命名をしている。また、tiny-youki も youki と同様に Rust 言語で実装した。なお、実装した tiny-youki のコードは、GitHub にて MIT ライセンスで公開している[2]。

2.1.1 tiny-youki 開発の背景

「tiny-youki」を開発する以前に、「tiny-runc」という非特権コンテナランタイムを開発していた。tiny-runc は Go 言語で実装しており、同じく OCI Runtime Specification に準拠した非特権コンテナランタイムを目指していた。この tiny-runc は、その名の通り、既存のコンテナランタイムである「runc」[3] を参考に実装している。この runc も、Go 言語で書かれたコンテナランタイムである。

しかし、Go 言語はグリーンスレッドモデルを採用しているため、プロセスとスレッドを高度に抽象化している。そのため、Go 言語でシステムコールやプロセス管理のための低レベル API を直接操作することが難しい。例えば、Go 言語では fork システムコールを直接操作することができず、常に fork と exec を同時に行うことしかできない。この制約により、親プロセスと子プロセスでの処理を分離して記述する場合、例えば子プロセス用の処理を親プロセスからサブコマンドとして呼び出すなどの回りくどい処理を行う必要が生じる。このアプローチでは、親プロセスで利用していた構造体やオブジェクトを子プロセスで再利用することが困難となるため、効率的なプログラミングが妨げられる。

例えば、runc ではこの問題を回避するため、fork を何度も行う必要のある名前空間の分離処理の部分を、C 言語で記述するという手法を採用している[3]。しかし、この手法は「シンプルで理解しやすいコードで簡潔に記述する」という tiny-runc の開発目標と相反するものである。

[2] https://github.com/n4mlz/tiny-youki
[3] https://github.com/opencontainers/runc/blob/main/libcontainer/nsenter/nsexec.c

このため、tiny-runc ではこの手法を採用せず、結果として複雑かつ非効率的な実装となる問題が生じた。

これらの課題を踏まえ、システムコールやプロセス管理において低レベルな制御を可能とし、かつ安全性とパフォーマンスを確保できる Rust 言語を選定し、tiny-youki の開発に活用した。Rust 言語を選択したことで、同じく Rust 言語で開発されている youki のコードベースを活用することが容易となり、結果として非特権コンテナランタイムの開発を効率的に進めることができた。

なお、tiny-runc も非特権コンテナランタイムとして動作するところまで実装しており、そのコードを GitHub にて MIT ライセンスで公開している[4]。

2.2 実装

2.2.1 名前空間の分離

tiny-youki では、名前空間の分離を行うことで、プロセス間の独立性を確保している。これは OCI Runtime Specification に明記されたものではないが、runc や youki などのコンテナランタイムで一般的に採用されている手法である。

tiny-youki では、以下の名前空間を分離している。名前空間の分離には、Linux カーネルの名前空間機能である `unshare` システムコールを利用している。

- User 名前空間: UID や GID、ケーパビリティの分離
- PID 名前空間: PID の分離
- UTS 名前空間: ホスト名の分離
- IPC 名前空間: SystemV IPC などのプロセス間通信の分離
- Mount 名前空間: マウントポイントの分離

2.2.2 OCI Runtime Specification への準拠

OCI Runtime Specification は、Open Container Initiative (OCI) によって定義された、コンテナランタイムの仕様ある。この仕様は、コンテナの実行環境を標準化し、異なるコンテナオーケストレーションツールやランタイム間での互換性を確保することを目的としている。実際は、OCI による runc の実装がデファクトスタンダードとなっている。

OCI Runtime Specification は、コンテナランタイムの動作を定義する JSON ファイルである `config.json` によってコンテナの設定を行う。OCI Runtime Specification は、この `config.json` が実際にどのように解釈されるか、どのような動作が期待されるかを定義している。

OCI Runtime Specification は GitHub にて公開されており[5]、同時に、Go 言語のパッケージとしても提供されている[6]。runc はこの Go 言語のパッケージを利用して、OCI Runtime Specification に準拠したコンテナランタイムを実装している。

他の言語で実装する場合は、そのスキーマやパーサーを自前で実装する必要がある。しかし、Rust 言語では、youki を開発している団体 が Rust 言語向けの OCI Runtime Specification のパーサーと構造体の定義を提供しているため、tiny-youki でもこれを利用することができた。

[4] https://github.com/n4mlz/tiny-runc
[5] https://github.com/opencontainers/runtime-spec
[6] https://pkg.go.dev/github.com/opencontainers/runtime-spec/specs-go

config.json は、コンテナのルートファイルシステムと共に、OCI Runtime Filesystem Bundle (以降、バンドル) としてディレクトリに配置される。runc や youki では、このバンドルを引数として受け取り、config.json を解釈してコンテナを起動する。tiny-youki でも同様に、バンドルを引数として受け取り、config.json を解釈してコンテナを起動するようにした。

```
use oci_spec::runtime::Spec;

...
pub fn new<P: AsRef<Path>>(bundle_path: P) -> Result<Self> {
    let bundle_path = bundle_path.as_ref();
    let config = Spec::load(bundle_path.join("config.json"))
        .map_err(|e| std::io::Error::new(std::io::ErrorKind::Other, e))?;

    Ok(ContainerBuilder {
        bundle_path: bundle_path.to_path_buf(),
        container: None,
        config,
    })
}
...
```

リスト 1: OCI Runtime Specification の `config.json` を解釈する処理
(tiny-youki/crates/libcontainer/src/builder/builder.rs からの抜粋)

2.2.3 非特権環境での動作

tiny-youki は、非特権環境で動作することを前提としている。unshare システムコールによる名前空間の分離には CAP_SYS_ADMIN ケーパビリティが必要であるが、tiny-youki ではこのケーパビリティを持たない状態からコンテナを起動する。

User 名前空間は、唯一 CAP_SYS_ADMIN ケーパビリティが不要な名前空間である。User 名前空間を分離することで、UID と GID のマッピングを行い、ホストの UID と GID をコンテナ内の UID と GID にマッピングすることができる。このとき、UID と GID を 0 にマッピングすることで、この名前空間内に閉じた root 権限を持つことができる。マッピングであるので、当然ながら名前空間内で作成したファイルやプロセスは、ホストからは元の UID と GID として見えることになる。

マッピングは、/proc/self/uid_map と /proc/self/gid_map に対して書き込むことで行うことができる。つまりは、unshare システムコールにより User 名前空間を分離した後に、元の名前空間 (ホスト) から /proc/self/uid_map と /proc/self/gid_map に対してマッピング情報を書き込むことになる。書き込みを行うまでは、分離した名前空間内のユーザーは全て nobody (UID: 65534, GID: 65534) として扱われる。この「nobody」は、Linux カーネルがデフォルトで持っているユーザーであり、特別な権限を持たない。なお、「65534」という値は、カーネルオプションの kernel.overflowuid、kernel.overflowgid の値である。

`/dev/pts/0`

```
$ unshare -U
$ whoami
nobody
$ echo $$
12345
$
```

`/dev/pts/1`

```
$ echo "0 1000 1" > /proc/12345/uid_map
$
```

`/dev/pts/0`

```
$ whoami
root
$
```

リスト 2: User 名前空間のマッピングの例

このマッピングでは、分離した名前空間内では root に見えているだけであって、実際には元の名前空間の root 権限は持っていない。もし User 名前空間内のプロセスに脆弱性があっても、 ホストの root 権限を奪われずに済むという利点があるが、マッピング元のホストのユーザーで行う権限のない操作を行うことはできない。しかし、名前空間内に閉じた `CAP_SYS_ADMIN` ケーパビリティを得ることができるので、他の名前空間を分離することができる。つまりは、User 名前空間を分離することで、他の名前空間を分離するための前提条件を満たすことができる。例えば、`CAP_SYS_ADMIN` を得たのちに Mount 名前空間を分離することで、コンテナ内でのファイルシステムのマウントを制御することができるようになる。

この方法の問題は、UID/GID をホストのものと 1:1 でしかマッピングできないことである。つまり、コンテナ内では root ユーザーしか使えないということである。nginx や mysql などのプロセスを動かす場合、これらのプロセスは root 以外のユーザーで動作する。このため、この方法ではこれらのプロセスを動かすことができない。

この問題を解決するために、`shadow utils` パッケージに含まれる `newuidmap` と `newgidmap` というツールを利用する方法がある。`useradd` コマンドでユーザーを追加すると、`/etc/subuid` と `/etc/subgid` にユーザー名と、そのユーザーに許可する subuid と subgid の範囲が記述される。`newuidmap` と `newgidmap` は、このファイルを読み込んで、指定された範囲内の UID と GID をマッピングする。`newuidmap` と `newgidmap` は、SUID ビットが付与されており、root 権限を持たないユーザーでも root 権限で実行される。このため、このツールを利用することで、コンテナ内で root 以外のユーザーを使うことができる。

```
$ cat /etc/subuid
user:100000:65536
$ newuidmap 23456 0 1000 1 1 100000 65536
$ cat /proc/23456/uid_map
         0       1000          1
         1     100000      65536
$
```

リスト 3: newuidmap による UID マッピングの例

上記の方法は、コンテナ内の UID 0 とホストの UID 1000 をマッピングし、コンテナ内の UID 1 65536 とホストの UID 100000 165535 をマッピングしている。この方法は、runc 及び youki でも採用されている方法である。tiny-youki でも、この方法を採用している。

2.2.4 コンテナのファイルシステムと State 構造体

コンテナのファイルシステムは、バンドル内の `rootfs` ディレクトリに配置されている。この `rootfs` ディレクトリは、コンテナのルートファイルシステムを表しており、コンテナ内からは `/` として参照される。この `rootfs` ディレクトリには、コンテナ内で利用するファイルやディレクトリが配置されている。例えば、`/bin` や `/lib` などのシステムファイルや、`/etc` などの設定ファイルが含まれる。

```
$ ls bundle/rootfs
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media opt  root  sbin  sys  usr
$
```

リスト 4: バンドル内の rootfs ディレクトリの例

この `rootfs` ディレクトリは、バンドル内の `config.json` に記述された `root` フィールドによって指定される。runc や youki では、この `rootfs` ディレクトリをマウントポイントとして指定し、overlayfs などのファイルシステムを利用して、コンテナのファイルシステムをオーバーレイすることで、コンテナのファイルシステムを構築している。しかし、この overlayfs は Mount 名前空間を分離していようと、ホストで root 権限を有していないと利用できない。runc では `fuse-overlayfs` というツールを利用して、非特権環境で overlayfs を利用する方法が提供されているが、この方法は非常に複雑であり、`fuse-overlayfs` を使用したプロセスが `pivot_root` による影響を受けないように、二度 Mount 名前空間を分離する必要がある。overlayfs を使用せず、コンテナ用のファイルシステムを別の場所にコピーしてバインドマウントを行うことで、多少のパフォーマンスの低下を覚悟すれば、同じ効果を得ることができる。tiny-youki では、シンプルな実装を目指すため、こちらの方法を採用した。

ファイルシステムの展開場所は定数として定義し、デフォルトで `/tmp/tiny-youki/containers/{container_id}/rootfs` に展開されるようにした。

```
1  pub const BASE_PATH: &str = "/tmp/tiny-youki";
2  pub const CONTAINER_PATH: &str = "containers";
```

リスト 5: tiny-youki/crates/libcontainer/src/constant.rs からの抜粋

```rust
pub fn new<P: AsRef<Path>>(container_id: &str, bundle_path: P) -> Result<Self> {
    let bundle_path = bundle_path.as_ref();

    let root = PathBuf::from(BASE_PATH)
        .join(CONTAINER_PATH)
        .join(container_id);

    if root.exists() {
        panic!("Container {} already exists", container_id);
    }

    let new_container = Container {
        state: State::new(container_id, bundle_path),
        root,
    };

    new_container.save()?;

    Ok(new_container)
}
```

リスト 6: tiny-youki/crates/libcontainer/src/container/container.rs からの抜粋

また、OCI Runtime Specification では、`state.json` というファイルにコンテナの状態を保存することが規定されている。この `state.json` は、コンテナの状態を説明する JSON ファイルであり、コンテナの `status` (`CREATED` や `RUNNING` など) や、プロセスの PID などが記述される。こちらの `state.json` も、OCI Runtime Specification によってスキーマが定義されている。tiny-youki ではこの `state.json` を、 `/tmp/tiny-youki/containers/{container_id}/state.json` に保存するようにした。

このように、コンテナごとに管理されるファイルやディレクトリが置かれる場所を用意しておくことは、youki でも採用されている方法である。例えば youki では、`/run/youki` がこれにあたる[7] 。

2.2.5 pivot_root によるルートファイルシステムの入れ替え

`chroot` と `pivot_root` の性質の違いを理解するために、本演習では tiny-youki とは別に、C 言語により簡単なコンテナを実装して両者を比較した。ここで検証に用いたコードは、GitHub にて公開している[8] 。

コンテナのファイルシステムを切り替える方法として一番簡単な方法は、`chroot` システムコールを利用する方法である。`chroot` システムコールは、指定されたディレクトリをルートディレクトリとして、プロセスのルートディレクトリを変更する。なお、プロセスがどのディレクトリをルートディレクトリとして扱っているかは、`/proc/self/root` というシンボリックリンクで確認できる。以下のコードは、C 言語による `chroot` システムコールを利用してコンテナを作成する例である。

[7] https://github.com/youki-dev/youki/blob/main/crates/libcontainer/src/container/container.rs#L28
[8] https://github.com/n4mlz/toy_mount_container

```
if (unshare(CLONE_NEWNS) == -1) {
    perror("unshare");
    return 1;
}

mount("none", "/", NULL, MS_REC | MS_PRIVATE, NULL);
mount(NEW_ROOT, NEW_ROOT, NULL, MS_BIND, NULL);

if (chroot(NEW_ROOT) == -1) {
    perror("chroot");
    return 1;
}
```

リスト 7: chroot システムコールを利用したコンテナの作成
(toy_mount_container/chroot_container.c からの抜粋)

この方法は非常に簡単であるが、セキュリティ上の問題がある。`chroot` は、ルートディレクトリを変更するだけで、カレントディレクトリは変更されない。そのため、`chroot` により作成されたコンテナでは、更にもう一度 `chroot` を行うことで、本来のルートディレクトリにアクセスすることができてしまう。例えば、`chroot` により `~/rootfs` をルートディレクトリとして設定した場合、`~/rootfs` は `/` として扱われる。この状態で、もう一度 `chroot /tmp` を行うことで、`~/rootfs/tmp` が `/` として扱われるようになる。しかし、カレントディレクトリは `~/rootfs` のままであるため、`chroot` 環境では本来たどり着けないはずのルートディレクトリの外にカレントディレクトリがあることになる。このような状態で例えば `cd ..` を行うと、ルートディレクトリの外に出ていかないかチェックが走るが、元々ルートディレクトリの外側にいる場合はこのチェックにマッチしないため、本来のルートディレクトリにアクセスできてしまう。以下のコードは、`chroot` により作成されたコンテナから脱獄する例である。

```
char pathname[PATHNAME_SIZE];
memset(pathname, '\0', PATHNAME_SIZE);

getcwd(pathname, PATHNAME_SIZE);
fprintf(stdout, "cwd: %s\n", pathname);

mkdir("test", 0);
chroot("test");

getcwd(pathname, PATHNAME_SIZE);
fprintf(stdout, "cwd: %s\n", pathname);

chdir("../../../../../../../../");

getcwd(pathname, PATHNAME_SIZE);
fprintf(stdout, "cwd: %s\n", pathname);
```

リスト 8: chroot により作成されたコンテナからの脱獄の例
(toy_mount_container/jailbreak.c からの抜粋)

この問題を解決するために、`pivot_root` システムコールを利用する方法がある。`pivot_root` システムコールは、プロセスのルートファイルシステム自体を入れ替えるものである。`pivot_root` システムコールは、現在のファイルシステムを別の場所にマウントし、新しいファ

イルシステムを / にマウントすることができる。また、pivot_root をした後に、元のファイルシステムをアンマウントすることもできる。この方法は、chroot と違い、ファイルシステム自体を入れ替えるため、chroot のように脱獄することができない。以下のコードは、C 言語による pivot_root システムコールを利用してコンテナを作成する例である。なお、pivot_root の引数として指定するディレクトリは、マウントポイントである必要がある。このため、pivot_root を行う前に、引数として指定したいディレクトリを自分自身にバインドマウントしている。

```
if (unshare(CLONE_NEWNS) == -1) {
    perror("unshare");
    return 1;
}

mount("none", "/", NULL, MS_REC | MS_PRIVATE, NULL);
mount(NEW_ROOT, NEW_ROOT, NULL, MS_BIND, NULL);

mkdir(NEW_ROOT "/old_root_fs", 0755);
if (syscall(SYS_pivot_root, NEW_ROOT, NEW_ROOT "/old_root_fs") == -1) {
    perror("pivot_root");
    return 1;
}
```

リスト 9: pivot_root システムコールを利用したコンテナの作成
(toy_mount_container/pivot_root_container.c からの抜粋)

この方法は、chroot に比べて複雑であるが、セキュリティ上の問題がないため、コンテナのファイルシステムを切り替える方法としては安全な方法である。

実際に挙動を確認するために、両者のコンテナを作成し、chroot により作成されたコンテナからの脱獄を試みる例と、pivot_root により作成されたコンテナからの脱獄を試みる例を実行した。結果は以下のようになった。

```
$ sudo ./chroot_container
/ # ./jailbreak
cwd: /
cwd: (unreachable)/home/user/container/chroot_fs
cwd: (unreachable)/
$
```

```
$ sudo ./pivot_root_container
/ # ./jailbreak
cwd: /
cwd: (unreachable)/
cwd: (unreachable)/
$
```

リスト 10: chroot により作成されたコンテナからの脱獄の例と、
pivot_root により作成されたコンテナからの脱獄の例

このように、chroot により作成されたコンテナからは、元のルートディレクトリにアクセスできてしまうが、pivot_root により作成されたコンテナからは、元のルートディレクトリにアクセスできないことが確認できた。

runc や youki では、この `pivot_root` を利用してコンテナのファイルシステムを切り替えている。tiny-youki でも、この `pivot_root` を利用してコンテナのファイルシステムを切り替えるようにした。tiny-youki では、以下のように `pivot_root` を行っている。

```
...
// recursively copy the rootfs from bundle to new_root
copy_dir_all(&self.bundle_root, &self.new_root)?;

// mount as private
mount::<str, str, str, str>(
    Some("none"),
    "/",
    None,
    MsFlags::MS_REC | MsFlags::MS_PRIVATE,
    None,
)?;

// mount rootfs
mount::<Path, Path, str, str>(
    Some(self.new_root.as_path()),
    self.new_root.as_path(),
    None,
    MsFlags::MS_BIND | MsFlags::MS_REC,
    None,
)?;

...

// mkdir old root
let old_root = self.new_root.join(".oldroot");
create_dir_all(&old_root)?;

// pivot root
pivot_root(self.new_root.as_path(), old_root.as_path())?;

// chdir to /
chdir("/")?;

// umount old root
umount2("/.oldroot", MntFlags::MNT_DETACH)?;

// remove old root
remove_dir("/.oldroot")?;
...
```

リスト 11: pivot_root によるルートファイルシステムの入れ替え
(tiny-youki/crates/libcontainer/src/namespaces/mount.rs からの抜粋)

また、`/proc` や `/dev`、`/dev/pts` などは、この `pivot_root` を行う前にマウントしておく必要がある。OCI Runtime Specification によると、これらマウントすべきディレクトリは `conig.json` に記述された `mount` フィールドによって指定される。tiny-youki では、この `mount` フィールドに記述されたマウントポイントを、`pivot_root` を行う前にマウントするようにした。

2.2.6 目的プロセス起動までのフロー

tiny-youki は、全体としては以下の 図 1 のようなフローで動作する。

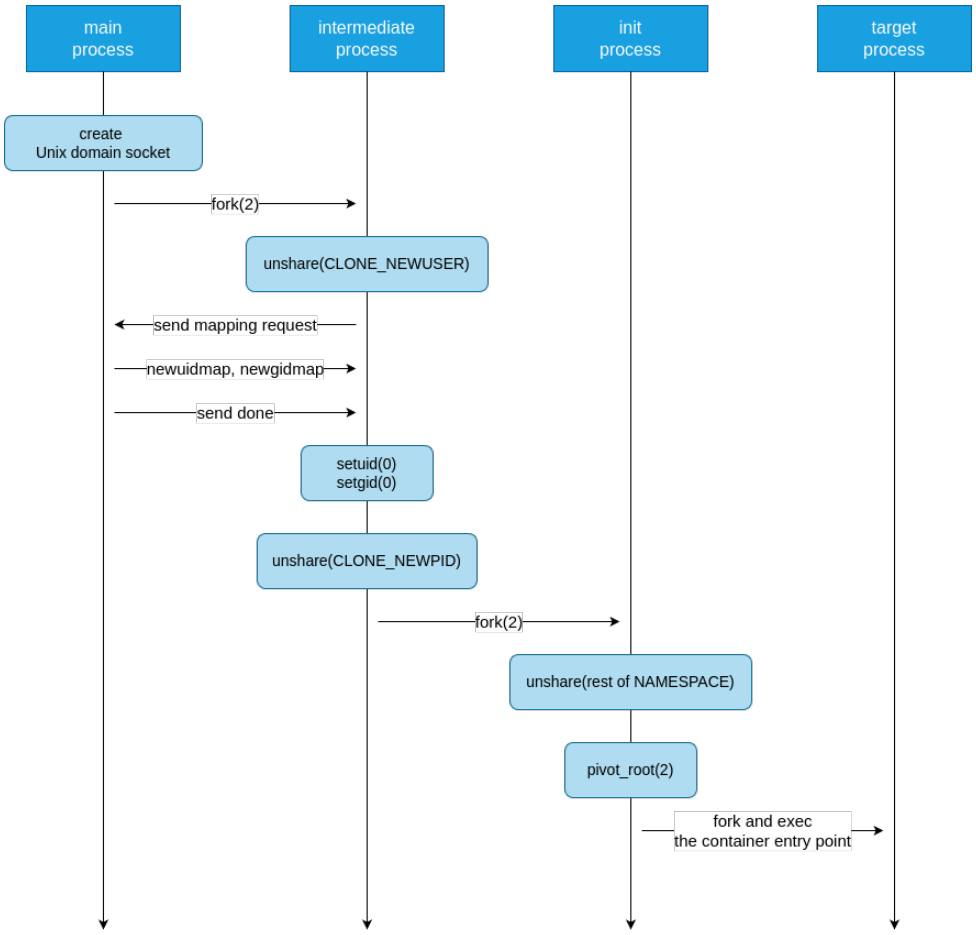

図 1: tiny-youki の目的プロセス起動までのフロー

ここで、`main process` は、`tiny-youki create` コマンドにより生成されるプロセスである。`intermediate process` は `main process` が生成したプロセスであり、Unix ドメインソケットによって `main process` と通信を行う。IPC 名前空間を分離すると、名前空間の内側と外側でのプロセス間通信に SystemV IPC や POSIX IPC を利用することができなくなる。そのため、runc や youki では、名前空間の内側と外側でのプロセス間通信を行うようなユースケースには Unix ドメインソケットを利用している[9]。tiny-youki でも、プロセス間通信には Unix ドメインソケットを利用した。図 1 の `main process` と `intermediate process` の通信ではまだ IPC 名前空間を分離していないので Unix ドメインソケットを利用する必要はないが、プロセス間通信に使用する媒体をプロジェクト内で統一するために、Unix ドメインソケットを利用している。

次に、`intermediate process` は、User 名前空間を分離する。この操作には `CAP_SYS_ADMIN` ケーパビリティが不要であるため、非特権で実行することができる。元の名前空間からマッピングを行うために、`intermediate process` は `main process` に、Unix ドメインソケットを使用してマッピングを依頼する。`main process` は、`intermediate process` からのリクエストを受け取り、マッピングを行う。このマッピングは、`newuidmap` と `newgidmap` を利用して行う。`intermediate process` は、マッピングが完了したことを `main process` から受け取ると、後の操作のために `setuid(0)` と `setgid(0)` を行い、`CAP_SYS_ADMIN` を得る。そして、PID 名前空間を分離する。

[9] https://github.com/youki-dev/youki/blob/main/crates/libcontainer/src/container/builder_impl.rs#L43

PID 名前空間を先に分離しておくのは、PID 名前空間を有効化するには一度 `fork` を挟む必要があるためである。

そして、`intermediate process` は `fork` を行い、`init process` を生成する。ここで言う `init process` は、全てのプロセスの親となる PID 1 のプロセスのことではなく、単にコンテナの初期化を行うプロセスのことである。このような役割を持つプロセスに対して `init process` という呼称を使うことは、runc や youki にも見られる[10]。図 1 の `init process` は、`config.json` に従って他の名前空間を分離し、ホスト名などの設定を行う。そして、コンテナ用のファイルシステムを用意し、`config.json` に従ってマウントを行う。そして、`pivot_root` を行い、コンテナのファイルシステムを入れ替える。最後に、`init process` は、`fork` と `exec` を行い、`target process` (目的プロセス) を起動する。

tiny-youki は、このように段階的に名前空間を分離していくことで、非特権環境でのコンテナの起動を実現している。

また 図 1 のようにコンテナランタイムの動作を説明する図は、youki の README にも記載されている[11]。

2.3 演習結果

以下に、演習終了時点での tiny-youki の動作のデモを示す。ArchLinux 上で tiny-runc により Ubuntu 22.04 のコンテナを起動し、ホスト名、ユーザ名、プロセスの PID 等がコンテナ内とホストで異なることを確認する。

[10] https://github.com/youki-dev/youki/blob/main/crates/libcontainer/src/process/container_init_process.rs

[11] https://github.com/youki-dev/youki?tab=readme-ov-file#-design-and-implementation-of-youki

```
$ whoami
host-user
$ id
uid=1000(host-user) gid=1000(host-user) groups=1000(host-user),998(wheel)
$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0  21936 12564 ?        Ss   Jan27   0:01 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Jan27   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Jan27   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/R-sync_wq]
root              6  0.0  0.0       0      0 ?          I<    Jan27    0:00 [kworker/
R-slub_flushwq]
root           7  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/R-netns]
...
$ uname -n
host
$ tiny-youki create some_container_id -b bundle
warning: unshare for cgroup and network has not yet been implemented
warning: mount for cgroup has not yet been implemented
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.4  0.0   6124  1524 ?        S    17:59   0:00 tiny-youki create
some_container_id -b bundle
root           2  0.0  0.0   2892  1760 ?        S    17:59   0:00 sh
root           4  0.0  0.0   7064  3000 ?        R+   18:00   0:00 ps aux
# uname -n
container
# exit
sh: 5: Cannot set tty process group (No such process)
$
```

リスト 12: tiny-youki の動作のデモ

このように、コンテナ内ではホストとは異なるユーザ名、ホスト名、PID が表示されていることが確認できた。

第 3 章　考察

本演習では、tiny-youki の実装を通じて、コンテナランタイムの基本的な仕組みを体系的に理解することができた。特に、コンテナ技術における非特権環境での動作の仕組みについて深く学ぶ機会となった。本プロジェクトでは、既存のコンテナランタイムである runc や youki を対象として、それらの何十万行にも及ぶ膨大なソースコードから必要な機能を特定し、その実装を追跡する過程で、コンテナランタイムの内部動作を詳細に理解することができた。この過程で、OSS の実装を読み解くスキルを向上させるとともに、大規模なコードベースを効率的に読み解く力も養うことができた。

また、OCI Runtime Specification のような標準仕様書を参照し、それに準拠した形で実装を行った経験は、標準仕様と実際のコードとの関係性を理解する上で非常に有益であった。仕様書の読み解き方や、それを具体的なコードへと落とし込むプロセスを経験する中で、設計段階での要件定義や仕様準拠の重要性についても改めて認識した。

特に、非特権環境でのコンテナランタイムの動作を実現するために必要なシステムコールやプロセス管理の知識を深めることができた点は、大きな成果である。例えば、`unshare` や `pivot_root` などのシステムコールの役割を理解するだけでなく、それらをどのように組み合わせることで非特権環境での動作を実現するかについて、実装を通じて具体的に学ぶことができた。

さらに、runc や youki の実装と比較しながら、tiny-youki のコードを小規模かつシンプルに保つことを目指した試みは、複雑なシステムの設計においてシンプルさを維持する難しさを学ぶ貴重な機会となった。この過程では、シンプルな設計を実現するために本当に必要であるコアのような処理がどこにあるかを捉え、どのように既存の実装から取捨選択を行い、必要な機能を最小限の形で再現するかという設計力を養うことができた。

総じて、本演習を通じて得られた知見は、コンテナ技術やシステムプログラミングに関する基礎的かつ応用的なスキルの両方に繋がるものであった。また、本プロジェクトで培った OSS の読解能力や仕様書準拠の設計・実装の経験は、今後のシステムソフトウェア開発においても有用であると考える。

第 4 章　今後の展望

tiny-youki は、基本的なリソース分離機能を備えているものの、OCI Runtime Specification で規定される `config.json` のすべてのフィールドをサポートしているわけではない。特に、ネットワーク設定やプロセスのリソース制限に関連する設定については、現時点では未実装である。本ランタイムは、実用性を追求することを目的としておらず、学習目的を第一に考えている。そのため、シンプルで理解しやすいミニマルなコードを目指し、OCI Runtime Specification の全機能を実装することは開発方針として想定していない。

しかしながら、ネットワーク設定やプロセスのリソース制限は、非特権環境におけるネットワーク設定やリソース制限の実現に関する実践的な知識を提供するものであり、学習目的としても非常に有用である。例えば、非特権環境でネットワーク設定を行う技術として、`slirp4netns` の利用が挙げられる。また、プロセスのリソース制限については、Linux の `cgroup` を活用することで実現可能である。これらの技術は、コンテナランタイムの理解をさらに深めるとともに、システムプログラミングにおける応用力を高める良い機会となる。そのため、今後の展望としては、これらの機能を tiny-youki に実装することを通じて、さらなる学びを得ることが挙げられる。

参考文献

[1] youki-dev, *youki*. [Online]. Available: https://github.com/youki-dev/youki

[2] Open Container Initiative, *OCI Runtime Specification*. [Online]. Available: https://github.com/opencontainers/runtime-spec

[3] Open Container Initiative, *runc*. [Online]. Available: https://github.com/opencontainers/runc