Linuxにおける仮想化技術を用いた世界OSの実装

石田 航[1]　新城 靖[1]　佐藤 聡[1]　中井 央[1]

概要：ソフトウェアの更新時に回帰テストを行うと、本番環境とテスト環境の差で動作が異なることがある。この環境の差を小さくし、回帰テストの効率を向上させることを目的として、本研究室では世界 OS という OS を開発している。世界 OS とはプロセスとファイルの容器を世界と定義し、その操作を可能とする OS である。世界の操作には、世界の生成、削除、融合、および融合内容表示の 4 種類が存在する。これらの操作を応用することによってファイル破壊からの迅速な回復や回帰テストの効率化が行える。既存の世界 OS では、融合時にユーザが派生ファイルを見逃してしまう問題や動作 OS が Solaris に制限されている問題が存在する。
本論文では Linux における仮想化技術を用いた世界 OS の実装について述べる。本実装では Linux Container と Another Union File System を組み合わせて多重継承を実現する。さらに、Linux Audit Daemon, およびその Linux カーネルの機能拡張で、コンテナごとのシステムコールの捕捉やコンテナ間でのプロセス移動を実現している。これにより、融合時のファイルの問題を解決し、Linux においても世界 OS の機能を利用可能にする。

1. はじめに

ソフトウェアを更新した際に、設定ファイルが破壊されることがある。こういった破壊がないことを回帰テストで確認することが重要となる。しかし、テスト用と本番用の環境を用意した場合、環境の差によって新たな問題が引き起こされることがある。

そのような問題を解決するために、トランザクション的な機能を持つオペレーティングシステムが開発されている [2]。本研究室においても、世界 OS と呼ぶオペレーティングシステムを開発している [3][4]。

従来のトランザクション的な機能を持つシステム [2][3][4] では、効率的なソフトウェアの更新およびテストを行うにあたってプログラムの実行環境を操作する。はじめに本番環境の複製を行って、テスト環境を作成する。次に、その中でテストを行う。テストに成功すればテスト環境を本番環境に融合し、問題があればテスト環境を削除する。ここでの環境は 1 つのトランザクションとみなせ、生成、融合、および削除はそれぞれトランザクションの Begin, Commit, および Abort とみなせる。

これらの従来のシステムには融合時のファイルの扱いに関する 2 つの問題が存在する。

- ユーザが派生ファイルの存在を見逃すことがある。例えば、図 1 では本番環境でソースファイルをコンパイルし、テスト環境でソースファイルのみを置き換えている。融合が起きた場合、古いオブジェクトファイルを見逃すことがある。
- 融合後にユーザが残したいファイルであっても上書きされて消失してしまう。例えば、図 2 では、テスト環境にパッチを当てたプログラムとその実行ログが残っている。融合する場合、従来のシステムではファイル全てが移動し、本番環境の実行ログがテスト環境の実行ログで上書きされてしまう。

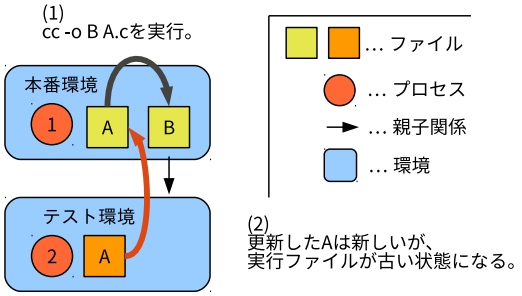

図 1　融合時における派生ファイルの見逃し

さらに、従来のシステムではベースとなる OS が Solaris または Mac OS X に制限されているという問題がある。

これらの問題に対して、本研究の目的を以下のように定める。

- 上で述べた融合時のファイルの扱いに関する 2 つの問題を解決する。

[1] 筑波大学
Uniersity of Tsukuba

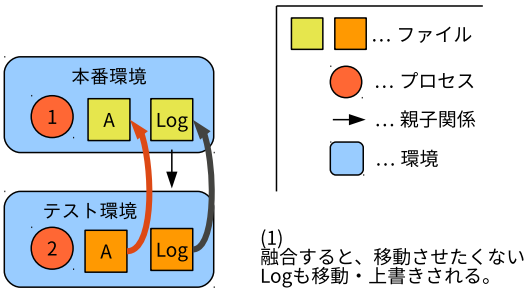

図 2 融合時における本番環境のログファイルの上書き

- 世界 OS を利用できるベースとなる OS を増やす。

本研究では、実装のベースとする OS として、広く用いられている OS の 1 つである Linux を用いる。これにより、Linux でも世界 OS が持つトランザクション的な機能が利用可能になる。また、実行環境の実装には仮想化技術の 1 つである Linux Container[5](以下、LXC) を用いる。さらに、生成時に行われるファイルシステムの継承は Another Union File System[6](以下、Aufs) を用いる。融合時のファイル問題を解決するために、システムコールの捕捉と解析を行う。これにより、利用者は問題の可能性のあるファイルを詳しく知ることが可能となる。捕捉には Linux Audit Daemon[7](以下、Auditd) を用いる。

最後に、本稿の構成について述べる。まず、2 章では、世界 OS について述べる。3 章では、コンテナを用いた世界 OS の設計について述べる。4 章および 5 章では、3 章で設計した世界 OS の実装について述べる。6 章では、その評価について述べる。7 章では、関連研究について述べる。最後に、8 章では、まとめと今後の課題について述べる。

2. 世界 OS

2.1 世界 OS の要件

世界 OS とは、計算機環境を世界としてとらえ、その操作を行う OS のことをさす。計算機環境である世界にはプロセスとファイルが含まれ、生成、削除、融合、および融合内容表示の 4 種類の基本操作が行える。世界 OS の要件は世界、および 4 種類の基本操作を実装する事である。

- 生成
  親世界を元に子世界を生成する。子世界は親世界のファイルをコピーオンライトによって継承する。生成が行われると、IP アドレスが割り当てられる。世界は多重継承が可能で、世界 OS によってつくられるシステム全体は図 3 のような有向非循環グラフ (Directed Acyclic Graph, 以下、DAG) となる。
- 削除
  指定した世界を削除し、プロセスの終了およびファイルの削除を行う。削除された世界が子世界を生成していた場合、その子世界も同様に削除される。
- 融合
  指定した世界同士を融合する。融合が実行されると、一方の世界のファイルとプロセスをすべてもう一方の世界へ移動する。同じ名前のファイルが存在すれば上書きされる。その後、空になった世界を削除する。
- 融合内容表示
  融合時にどのようなファイルの追加・削除が行われるかを表示する。利用者は、その表示をもとに予期せぬファイルの上書きがないかを確認できる。

これら 4 種類の操作を応用することによって、ファイル破壊からの回復やソフトウェアの回帰テストを行うことができる。

2.2 世界 OS のコマンド

本節では、世界 OS を用いて WordPress を導入する例を用いて、世界 OS のコマンドを説明する。はじめに Parent という世界で変更前の WordPress を HTTP サーバ上で動作させる。それには次のように wexec コマンドを用いる。

```
$ wexec Parent service apache2 start
```

また、複数のコマンドを対話的に実行したい場合には、次のように wconsole コマンドを用いることによって擬似端末に接続してコマンドを実行する。

```
$ wconsole Parent

Parent login: ubuntu
Password:
Last login: Mon Sep 14 09:00:44 JST 2015 on
    lxc/console

$ emacs /etc/apache2/ports.conf

...

$ service apache2 start
$ exit
```

次に世界の**生成**を行う。ここでは Child1, および Child2 の 2 つの世界を生成する。それには、次のように wcreate コマンドを用いる。

```
$ wcreate Child1 Parent
$ wcreate Child2 Parent
```

次に Child1 では WordPress のアップデートを行い、Child2 で MySQL のアップデートや再起動を行う。

```
$ wexec Child1 -- patch /home/ubuntu/html/
    wordpress < wordpress.patch
$ wexec Child2 service mysql stop
$ wexec Child2 dpkg -i /home/ubuntu/new-mysql.
    deb
```

次に世界の**生成**で Child1, および Child2 を多重継承した Child3 という世界を生成する。それには wcreate コマンドの 2 引数目以降に、親世界を複数指定する。

```
$ wcreate Child3 Child1 Child2
```

次に、Child3 でテストを実施する。また、外部から接続できるように wip コマンドを用いて、Child3 の 80 番ポー

トをホスト OS の 50088 ポートにフォワーディングしている。この一連の動作で作られる世界の関連は図 3 のようになる。

```
$ wexec Child3 service apache2 start
$ wip Child3 forward 80 50088
```

テストに失敗すれば世界の**削除**を行って Child3 を消去し、Parent で動作している WordPress を動作させたままにする。wdelete コマンドによって世界の**削除**を行う。

```
$ wdelete Child3
```

テストに成功し、世界の**融合**の可否の見当をつけるために、世界の**融合内容表示**を行う。それには次のように、wdiff コマンドを用いる。この表示で使われる記号を表 1 に示す。

```
$ wdiff Child3 Parent
World: Child3 -> Parent

! /var/log/apache2/access.log
+ /home/ubuntu/html/wordpress/wp-content/wp-
    config.php
...
```

このとき、wset コマンドを使うことにより、融合の対象から外すことができる。以下の例では、テスト環境のログファイルを融合の対象から外している。

```
$ wset Child3 remove /var/log/apache2/access.
    log

$ wdiff Child3 Parent
World: Child3 -> Parent
+ /home/ubuntu/html/wordpress/wp-content/wp-
    config.php
...
```

融合内容表示で問題がないと判断した場合、または許容できる場合は世界の**融合**を行う。この時、Child1, Child2, および Child3 のプロセスとファイルが Parent に移動し、Child に対して世界の**削除**が行われる。

```
$ wmerge Child1 Parent
$ wmerge Child2 Parent
$ wmerge Child3 Parent
```

2.3 世界 OS におけるトランザクション分離レベル

オペレーティングシステムでトランザクション的な仕組みを実装することの有用性は古くから試みられている [1]。この仕組みの実装方法として、プログラムの実行環境を複製するという手法がある。例えば、Spinellis によるもの [2] ではファイルシステムの複製を行っている。既存の世界

表 1 融合内容表示コマンド (wdiff) で使われる記号

表示	内容
!	親の書き込みが失われる。
?	融合前の派生物が存在する。
+	ファイルが追加される。
-	ファイルが消去される。

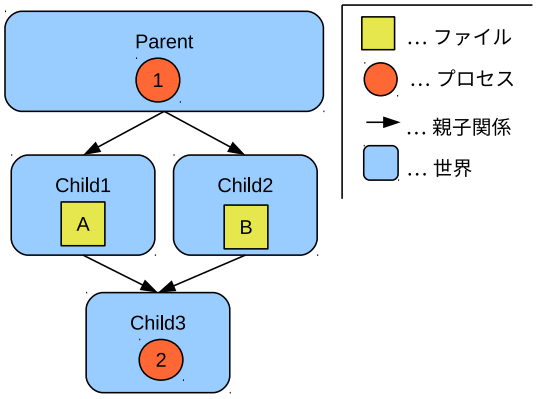

図 3 世界の生成および多重継承のイメージ

OS も同様の手法を用いており、inode に似た構造体を定義し参照するファイルのブロックを環境ごとに保存するもの [3]、システムコールの引数を改変するもの [4] が存在する。

複数トランザクションを並行して動作させ、どれだけデータの一貫性が保たれるかの程度をトランザクション分離レベルと呼ぶ。最も高い分離レベルをもつものとして、直列化可能性 (Serializable) があり、LOCUS[1] では、直列化可能性を満たしている。世界 OS 中の世界をトランザクションとみなしたとき、既存の世界 OS[3][4] では直列化可能性を満たすことを求めておらず、分離レベルは低い。理由としては、世界 OS の主な用途の 1 つであるソフトウェアの更新を行うにあたり、直列化可能性の条件を満たさなくても問題がないということがあげられる。2.1 節で述べた WordPress のテストがその具体例にあたり、融合前に Child2 中の PHP 関連のファイルが子世界である Child3 から読み込まれたとしても問題はない。本研究でも、既存の世界 OS[3][4] と同様、ソフトウェアの回帰テストを主な用途とし、直列化可能性を満たさなくてもよいとする。

2.4 既存の世界 OS の実装

既存の世界 OS として、Jun[3] らによるものと石井 [4] らによるものがある。Jun らによる実装では、分散ファイルシステムおよび RPC を用いて世界 OS のファイルサーバを実現している。また、石井らによる実装では、プロセストレースによってシステムコールの引数を書き換えることで世界 OS の実現を行っている。

Jun らによる実装と比較すると、本研究の実装ではアプリケーションを RPC を用いるように書き換える必要がなく、プロセス移動が行えるという利点がある。石井らによる実装と比較すると、本研究での実装は 1 章で述べた融合時のファイル問題を解決し、部分的な融合が行えるという利点がある。

3. コンテナを用いた世界 OS の設計

本研究では、Linux における仮想化技術と既存ソフトウェ

アを再利用して世界 OS を実装する。この章では、LXC, Aufs および Auditd を用いた世界 OS の設計について述べる。

3.1 コンテナによる世界の実装

本研究では、世界の実装に仮想化技術の 1 つであるコンテナを用いる。コンテナはホスト OS から隔離された計算機環境のことである。LXC[5] は Linux カーネルが提供しているネームスペースという機能を用いて実現がなされている。LXC では、コンテナ同士はカーネルを共有しつつも PID、ホスト名、およびファイルシステム等をコンテナごとに隔離することが可能となっている。応用例として、コンテナと chroot を組み合わせ、本体とは別のパッケージ管理システム等を導入することがある。

LXC におけるコンテナの特徴は、プロセスとファイルの入れ物という世界の性質を満たす。したがって、コンテナを世界として利用することができる。本研究においては、世界の生成や削除にコンテナの生成および削除を行う。

既存の世界 OS[3][4] と比較したとき、コンテナを再利用することによって、既存の世界 OS で行われていた隔離の実装を省略できるという利点がある。また、仮想化技術の 1 つである仮想計算機と比較すると、コンテナはホスト OS のファイルシステムやプロセスなどを共用している。そのため、世界の基本操作の実装が簡易に行えるという利点がある。例えば、融合におけるファイル移動やプロセス移動はホスト OS 上の資源の操作で実現できる。

3.2 Aufs によるコピーオンライトの実現

Aufs[6] とはユニオンファイルシステムの 1 種である。これを利用して、複数ファイルシステムを重ねあわせて単一のディレクトリとして扱うことができる。LiveCD では読み書き可能なメモリファイルシステムと読み込み専用の CDROM のファイルシステムをユニオンマウントすることで、あたかも CDROM 上のファイルに書き込みが行えているように見せかけている。

また、Aufs による、世界のファイルシステムの設定の手順は次のようになる。

(1) 親世界のコンテナのルートディレクトリの上に Aufs により書き込み可能なディレクトリを重ねあわせ、それを指定したディレクトリにマウントする。
(2) マウントされたディレクトリを子世界のコンテナのルートとする。

本研究では、(1) の書き込み可能なディレクトリを**世界の中身のディレクトリ**と呼ぶ。(2) のディレクトリを**マウントディレクトリ**と呼ぶ。図 3 中の Child1, 2, および 3 のファイルシステムの全体像を図 4 に示す。この図では、Child1 からはファイル A, Child2 と Child3 からはファイル A, およびファイル B がそれぞれ見えている。個別の世界の中身のディレクトリは書き込み可能となっている。さらに、親子間でファイル追加・削除の影響を受ける。例えば、図 4 において、Child1 中のファイル A が削除されれば、同時に Child2, および Child3 からファイル A が削除される。Child2 中でファイル A が削除されると、ホワイトアウトと呼ばれるファイルが Child2 の世界の中身のディレクトリに生成される。その結果、Child2, および Child3 からファイル A が参照できなくなる。

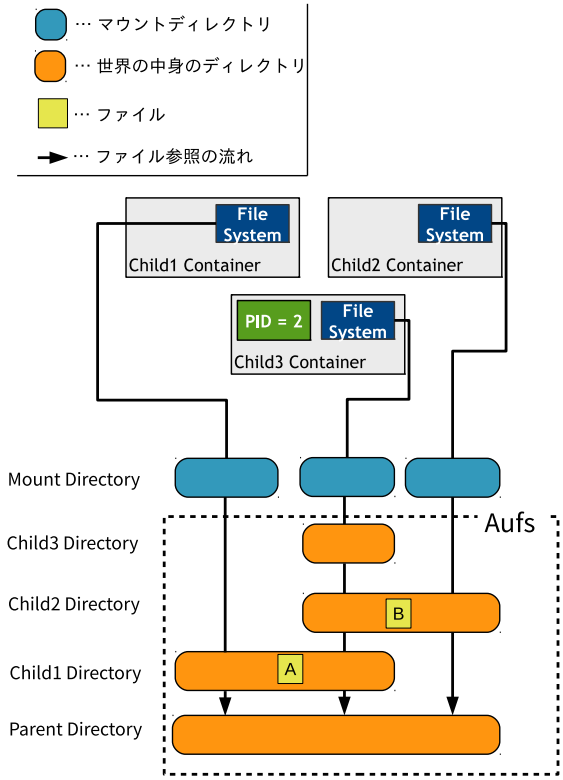

図 4 Child1, 2, および 3 を Aufs によって実現した例

3.3 Auditd によるファイルアクセスの監視

Auditd[7] はシステムの監査を行うためのプログラムであり、システムコールの捕捉を行う。Auditd はユーザ空間で動作するプログラムとカーネル空間で 2 つのプログラムから構成されており、カーネル空間ではシステムコールの捕捉を行い、ユーザ空間ではその結果を受信している。その用途としては、プロセスの異常動作を発見したり、システムへの侵入検知を行うために用いられる。

本研究ではこの Auditd を、世界の中身のディレクトリで行われるファイルの読み込みと書き込みの捕捉に用いる。これらの捕捉によって、1 章で述べた 2 つの問題を解決する。また、本研究では Auditd の機能を拡張している。その拡張内容については 5.1 節で述べる。

3.4 世界を操作するプログラムとユーザインタフェースの設計

本研究で実現する世界 OS はユーザインタフェースとし

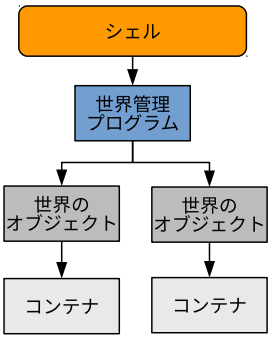

図 5 プログラムの全体像および命令の流れ

てシェルを持ち、世界を管理するための**世界管理プログラム**を持つ (図 5)。シェルはユーザからコマンドの入力を受けとり、解析を行った後、世界管理プログラムに命令を送信する。

世界管理プログラムは世界の基本操作を行い、親子関係を管理する。このプログラムは次の要素を保持する。

世界のリスト
: 世界のオブジェクトを保存するリスト。

衝突管理配列
: 世界間でファイルの読み書きの衝突を検知するための連想配列。

キュー
: 読み書きを補足した後、その結果の一時保存を行うキュー。

世界の基本操作が行われると、世界に関する資源を保存するオブジェクトの操作が行われる。本研究ではこのオブジェクトを**世界のオブジェクト**と呼ぶ。世界のオブジェクトは主に次の要素で構成されている。

コンテナ
: LXC のコンテナ。

世界の名前
: 世界につける名前および世界に所属するコンテナのホスト名。

ID
: UUID(Universal Unique Indentifier) で定義される、世界を一意に表す識別子。

PID リスト
: 世界の中で実行されているプロセスの PID のリスト。

各種パス
: **世界の中身のディレクトリ**や**マウントディレクトリ**へのパス。

親世界のリスト
: 生成元の世界への参照。多重継承が行えるため、複数持つことが可能。

4. ユーザ空間で動作する世界 OS のプログラム

この章では 3 章で述べた設計に基づいた、ユーザ空間上で動作するプログラムについて述べる。本研究では世界 OS におけるユーザインタフェース、世界管理プログラムおよび基本操作の大部分を Python で記述し、基本操作の一部を C 言語で記述した。

4.1 世界の DAG の管理

世界の操作を行う際、世界の DAG を管理する必要がある。例えば、図 4 で Child1 を削除した場合、Child1 および Child3 は削除される。また、Child3 は Child1, および Child2 を多重継承しているので、Child3 のファイルシステムを設定するとき、その親をたどってユニオンマウントを行う。この時、各々の世界の親子関係を調べる必要がある。

本研究では、世界のオブジェクトの親世界のリストによって世界の DAG を実現する。削除を行った場合、指定した世界への参照を持つ子世界を幅優先探索によって探索する。一方、生成した子世界のユニオンマウントを行う場合、トポロジカルソートによって子から親へ並び替えを行い、順番にマウントを行う。

4.2 システムコールの捕捉

世界の融合内容表示では、世界に所属するプロセスが読み書きを行ったファイルを表示する。その内容は Auditd のシステムコールの捕捉機能によって取得する。

読み書きの衝突検出のために、read, および write システムコールの捕捉を行うことが考えられる。しかし、それらは呼び出される頻度が多いシステムコールであるため、これらの捕捉を行うとシステム全体のオーバーヘッドが大きくなってしまう。本研究の実装では、オーバーヘッドの軽減のため open, および openat システムコール [*1] のフラグを捕捉することによって読み書きの判断を行った。例えば、O_RDONLY, O_WRONLY, および O_RDWR のフラグはそれぞれ読み込み, 書き込み, および書き込みに対応する。

世界 OS のシェルが起動すると、Auditd は次のようにシステムコールの捕捉と解析を行う。ログの捕捉と解析はスレッドによってなされる。スレッドは取得用、解析用、および反映用の 3 種類がある。ログを解析するための一時的な保存場所にキューが用いられる。また、キューからログを取り出したとき、ログは JSON 形式に整形される (図 6)。

(1) 監視するシステムコールと監視するディレクトリを登録する。登録するディレクトリは**世界の中身のディレクトリ**となる。
(2) コンテナ中でシステムコールが発行される。
(3) Auditd がシステムコールを補足し、データを指定し

[*1] openat システムコールは open システムコールと同等の働きを行うが、引数が異なる。具体的にはディレクトリのファイルディスクリプタおよび、そのディレクトリからの相対パスの文字列を取る。

```
# Before
node=experiment type=SYSCALL msg=audit
    (1438768306.227:471):  ...  key="test"
# After
{'node':'experiment', 'type':'SYSCALL', 'msg':
    'audit(1438768306.227:471):', ...  , 'key'
    :test"}
```

図 6 Auditd ログの整形

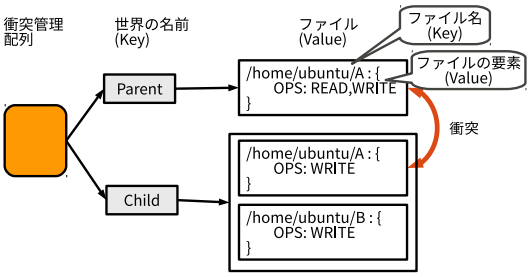

図 7 衝突管理配列の中身

た Unix ドメインソケットに転送する。

(4) 世界管理プログラムの取得用スレッドが Unix ドメインソケットからそのデータを受け取り、キューに挿入する。

(5) 世界管理プログラムの解析用スレッドがキューのログを整形する。

(6) 世界管理プログラムの反映用スレッドが整形されたログを衝突管理配列に反映する。

衝突管理配列は連想配列となっており、世界の名前をキーとし、アクセスが行われたファイルのデータを値としている。また、その中にさらなる連想配列がデータ中に存在し、その中ではファイル名をキーとし、そのファイルのアクセス方法を値としている (図 7)。

図 7 では Parent と Child の 2 つの世界で、/home/ubuntu/a や/home/ubuntu/b がそれぞれアクセスされた場合の衝突管理配列を示している。また、OPS(Operations) フィールドには読み・書きといったファイルアクセスの種類を保存する。

4.3 世界の生成

世界の生成はコンテナの生成および、**世界の中身のディレクトリ**と**マウントディレクトリ**の設定を通して行われる。世界の生成を実行すると、はじめに親の LXC のコンテナを複製する。次に、世界のマウントディレクトリをコンテナのルートディレクトリとする。この操作は LXC の設定ファイルを書き変えることで実現する。なお、世界の IP アドレスの設定は LXC により自動的に行われる。世界の生成は、以下に示すコマンドと同じような動作を行う。なお、これらのコマンドは説明のために示したものであり、実際は Python で記述されたプログラムで実行される。

```
# /mnt_child ... childのマウントディレクトリ
# /root_child ... childのルートディレクトリ
# /var/lib/lxc/child/config ...
    childのコンテナの設定ファイル

lxc-clone -o parent -n child
mkdir /mnt_child /root_child
mount -t aufs -o br:/root_child \
   /mnt_child
mount -o remount,append:/root_parent \
   /mnt_child
sed -i 's:/var/lib/lxc/child/rootfs: \
   /mnt_chlid:g' \
   /var/lib/lxc/child/config
lxc-start -n child -d
```

4.4 世界の削除

世界の削除を行うと、世界の DAG に基づき、対象となる世界と関連するコンテナおよび**世界の中身のディレクトリ**や**マウントディレクトリ**をすべて削除する。また、ホスト名やコンテナ自体の情報が記述されたファイルを保存するディレクトリも同様に削除する。世界の削除は、以下に示すコマンドと同じような動作を行う。

```
# /root_child ...
    childの世界の中身のディレクトリ
# /mnt_child ... childのマウントディレクトリ
# /var/lib/lxc/child ...
    childのコンテナ情報を保存したディレクトリ

lxc-stop -n child
umount /mnt_child
rm -rf /mnt_child /root_child \
   /var/lib/lxc/child
```

4.5 世界の融合

世界の融合を行うと、コンテナ間でファイルおよびプロセスの移動を行い、世界のデータ構造の情報を移動先の世界が受け継ぐ。ファイル移動については、世界の中身のディレクトリにあるファイルを移動することによって実現する。その後、ファイルのホワイトアウトの検索も行い、ホワイトアウトと対応するファイルが親世界の世界の中身のディレクトリに存在すれば、そのファイルを削除する。これらの処理は、以下に示すコマンドと同じような動作を行う。

```
# .wh.A ... Aのホワイトアウト
# root_child ...
    childの世界の中身のディレクトリ

mv /root_child/* /root_parent/

for f in /root_child/.wh.*
do
   g=`basename $f | sed 's:\.wh\.::'`

   if [ -f /root_parent/$g ];
   then
     rm /root_parent/$g;
   fi
done
```

プロセス移動については、対象となるプロセスのルートディレクトリとカレントディレクトリ、およびネームス

ペースをそれぞれ書き換えることで実現する。この書き換えにあたり、本研究ではシステムコールの追加を行った。その実装の詳細については 5.2 節で述べる。

```
# PID=1000のプロセスのルートを
   /mnt_parentに変更する
chroot2 /mnt_parent 1000

# PID=1000のプロセスのカレントディレクトリを/
    mnt_parentに変更する
chdir2  /mnt_parent 1000

# PID=1000のプロセスのネームスペースをPID=456
    を持つプロセスのものに書き換える
setns2  1000 456
```

4.6 世界の融合内容表示

融合内容表示を行うと、衝突判定と結果表示が行われる。衝突判定は、図 7 における衝突管理配列中の OPS フィールドを用いて判定する。判定結果は衝突、警告、ファイル追加の 3 つに分けられる。この 3 種類の判定の例を表 2 に示す。表においては、アクセスなしを"-", 読み込みを R, 書き込みを W とする。

衝突は、融合が起こった際に移動先のファイルが消失する状態を示す。警告は、融合を行った場合にファイルの派生物が存在する状態を示す。ファイル追加は、移動先に同名のファイルが存在せず、ファイルが追加されることを意味する。

融合内容表示においては、衝突判定の結果をファイルごとに表示する。また、衝突判定の結果とは別に、ファイル削除の結果も表示する。ファイル削除の結果は世界の中身のディレクトリにおけるファイルのホワイトアウトから求められる。

5. カーネル空間で動作する世界 OS のプログラム

本研究では Linux カーネルの改変を行うことにより、ネームスペース単位でのシステムコール捕捉やネームスペースの書き換えを実現している。この章では、それらの実装について述べる。

5.1 Auditd の改変

実装で用いている Linux-3.13.11 上の Auditd によるシステムコールの捕捉はネームスペース単位では行えない。すなわち、どのコンテナがどのシステムコールを実行したかを知ることができない。そのため、本研究では Auditd で出力されるログにネームスペースの情報を追加した。

表 2 ファイルアクセスの衝突判定の例

親世界	子世界	衝突判定
W	W	衝突
W	R	警告
R	W	警告
-	W	ファイル追加
W	-	ファイル追加

```
node=experiment type=SYSCALL msg=audit
    (1438768306.227:471): arch=c000003e
    syscall=4 ns=ffffffff81c485a0
ns_name=root ...
```

図 8 拡張した Auditd の出力例

Linux カーネルにおいて、プロセスを表す構造体に task_struct という構造体が存在する。この task_struct の中にネームスペースを管理する識別子 (ns_proxy 構造体へのポインタ) が存在する。本研究では、ns_proxy 構造体の情報を取得し、Audit ログの出力時にその情報を挿入する。その出力例を図 8 に示す。このログにおいて、ns が ns_proxy へのポインタに対応し、ns_name がコンテナのホスト名に対応する。

5.2 システムコールの追加

世界間でプロセス移動を行うにあたり、本研究では既存のシステムコールに似せたシステムコールを追加している。実装にあたって参考にしたシステムコールは chroot, chdir, および setns の 3 つである。本研究のこの 3 つのシステムコールで、任意のプロセスを操作できるように拡張した。

まず、chroot を拡張した chroot2 について説明する (図 9)。chroot からの主な変更点は引数部分および、図 9 の 6, および 12 行目にあたる。まず、引数には chroot のものに加え、変更先の PID を取るように変更した。6 行目の find_task_by_vpid 関数は chroot2 の引数の PID に対応した task_struct 構造体を取得する関数である。ケーパビリティやパーミッションに関連した条件分岐を経て、12 行目の set_fs_root で設定で実際にプロセスのルートディレクトリの変更を行う。通常の chroot では現在実行しているプロセスの task_struct をさす、current マクロを取る。この current マクロの代わりに、find_task_by_vpid で取得した task_struct 構造体を引数にとることにより、別プロセスのルートディレクトリを変更することができる。

次に、chdir を拡張した chdir2 について説明する (図 10)。chdir では、set_fs_pwd を実行することによってプロセスのカレントディレクトリを変更することができる。chroot2 と同様に、set_fs_pwd の引数を current マクロから指定したプロセスの task_struct に変更することにより、任意のプロセスのカレントディレクトリを変更する。

次に、setns を拡張した setns2 について説明する (図 11)。世界の**融合**によってプロセスが移動した際、プロセスが所属する PID ネームスペースを移動先の PID ネームスペースに書き換える必要がある。

本研究では find_task_by_vpid を用いて setns2 を実装することで、任意のプロセスのネームスペースを書き換える。

```
1  SYSCALL_DEFINE2(chroot2, const char __user *,
       filename,pid_t, vpid)
2  {
3    struct path path;
4    int error;
5    unsigned int lookup_flags = LOOKUP_FOLLOW |
       LOOKUP_DIRECTORY;
6    struct task_struct *task = find_task_by_vpid
       (vpid); // 追加
7    if(!task){
8      return -EINVAL;
9    }
10
11 ...
12   set_fs_root(task->fs, &path);//current->
       fsからtask->fsに変更
13   error = 0;
14 ...
15 }
```

図 9 chroot2 の実装の一部

```
1  SYSCALL_DEFINE2(chdir2, const char __user *,
       filename,pid_t, vpid)
2  {
3    struct path path;
4    int error;
5    unsigned int lookup_flags = LOOKUP_FOLLOW |
       LOOKUP_DIRECTORY;
6
7    struct task_struct *task = find_task_by_vpid
       (vpid);// 追加
8
9  ...
10
11   set_fs_pwd(task->fs, &path);//current->
       fsからtask->fsに変更
12
13 ...
14
15 }
```

図 10 chdir2 の実装の一部

この書き換えにより、ネットワークネームスペースとホストネームスペースはその結果が即時に反映される。ネットワークネームスペースを切り換えることで、コンテナの IP アドレスが変化する。移動するプロセスがコネクションを張っている場合、その接続は維持される。例えば、融合される世界で SSH でログインしていた場合、接続はそのままになる。しかし、PID ネームスペースは結果が反映されない。結果が反映されるのは、指定したプロセスの子プロセスからとなる。

この問題の解決のため、fork で実行される PID ネームスペース書き換えの処理をシステムコール中で実行している。この処理の手順は、次のようになる (図 11)。

(1) 引数 vpid で指定したプロセスとそのネームスペースの情報を得る (12 行目から 24 行目)
(2) 所属していたネームスペースから指定したプロセスの情報を削除する (26 行目)
(3) PID から得られるネームスペースを指定したプロセスに適用する (28 行目から 33 行目)

```
1
2  SYSCALL_DEFINE3(setns2, int, fd, int, nstype,
       pid_t, vpid)
3  {
4  ...
5          struct pid *pid_tsk;
6          struct pid_namespace *tmp,*pid_ns;
7
8          struct upid *upid;
9          struct pid_map *map;
10         int i;
11 ...
12         pid_tsk=get_task_pid(tsk,PIDTYPE_PID);
13 ...
14         if((ops->type && CLONE_NEWPID) &&
               nstype){
15
16           if(pid_tsk == NULL){
17             goto out;
18           }
19
20           tmp = (struct pid_namespace*)(ei->ns
                 );
21           pid_ns = tmp;
22           pid_tsk->level = tmp->level;
23
24           upid = pid_tsk->numbers + tmp->level
                 ;
25
26           free_pidmap(upid);
27
28           for(i = pid_tsk->level; i>=0; i--){
29             pid_tsk->numbers[i].ns = tmp;
30             if(tmp->parent == NULL){
31               break;
32             }
33             tmp=tmp->parent;
34           }
35         }
36
37 ...
38 }
```

図 11 setns2 の実装の一部

6. 評価

6.1 機能評価

1 章で述べたとおり、既存のシステムでは、派生ファイルが見逃されたり、上書きの意図がないファイルが融合によって上書きされてしまうという 2 つの問題がある。本研究ではこれらの問題をファイルアクセスの捕捉によって解決する。

派生ファイルを見逃す問題は、ファイルアクセスされたファイルの絶対パスを記録することで解決する。利用者は 2.1 節で示した融合内容表示によって、派生ファイルを確認できる。

意図しないファイルの上書き問題は、2.1 節で示した wset コマンドを用い、融合の対象から外すことで解決する。この操作によって、特定ファイルの上書きを避け、部分的な世界の融合が実現できる。

現在の実装では、ファイルの読み書きを追跡するために open, および openat システムコールを追跡している。

```
$ wexec root -- tar xzf /home/ubuntu/linux
    -2.6.32.tar.gz -C /home/ubuntu/
$ wcreate child root
$ wexec child -- bash /home/ubuntu/patch.sh
$ wdiff child root
# 以下のいずれかを実行する。
# $ wmerge child root
# $ wdelete child root
$ wexec root -- rm -rf /home/ubuntu/linux
    -2.6.32
```

図 12 実験 1 で用いるコマンド

```
#!/bin/bash
patch -f -p1 -s -d /home/ubuntu/linux-2.6.32 <
     /home/ubuntu/patch-2.6.32.67
```

図 13 patch.sh の中身

mkdir に関しては追跡を行っていないが、Aufs によって世界の中身のディレクトリを起点としたパス名を保持している。そのため、ディレクトリ作成を含めたファイルの融合を実現できる。しかし、現在の実装ではディレクトリの改名が行えない。改名が行われると、ディレクトリの作成と同様に扱われる。また、Aufs の機能により、ファイルの改名はコピーと削除として扱われる。

6.2 実験の内容

性能評価を行うために、次の 2 つの実験を行う。

実験 1 基本操作を用いたパッチ当てと元ファイルの回復を行う。各基本操作の実行時間を測定する。

実験 2 パッチ当てを行う。Auditd, および Aufs が性能に及ぼす影響がどれだけかを調べる。

実験 1 では Linux カーネルソースツリー、およびテキストデータのパッチ当てを行い、最後に世界の融合または削除を行う。融合は削除の操作を含むため、削除を行う場合と融合を行う場合の 2 種類に分け、基本操作ごとの実行時間を計測する。ここでは、wcreate, wdelete, wmerge, および wdiff コマンドの実行時間で測定する。実験は連続で 10 回行い、実行時間の平均値を求めた。また、場合分けによって 2 重に取得できる wcreate, および wdiff は wmerge を用いた実験の数値を用いる。

実験 1 で用いるコマンドでは、はじめに親世界でカーネルソースを展開し、次に子世界 child, および child2 を作り、child でパッチ当てを行っている (図 12)。

図 12 における patch.sh の中身を図 13 に示す。パッチを当てる Linux カーネルソースツリーのバージョンは 2.6.32 であり、パッチは patch-2.6.32.67(サイズ: 4.7MB, 適用ファイル数: 2511 ファイル) である。GNU patch のバージョンは 2.7.1 である。

実験 2 では図 13 のシェルスクリプトを実行し、そのパッチ当ての時間を計測する。実行時間においては、実験 1 と同様に 10 回行い、その平均値を求める。また、実験の条件は Auditd, および Aufs の組み合わせとなる。その条件を以下に示す。なお、() の場合は LXC で実行した状態と等価である。

(1) システムコールの捕捉を行わない状態かつ世界 OS 上での親世界で実行 (Aufs, Auditd ともに無効)
(2) 世界 OS 上での親世界で実行 (Aufs が無効、Auditd が有効)
(3) 世界 OS 上での子世界で実行 (Aufs が有効、Auditd が無効)
(4) 世界 OS 上での子世界で実行 (Aufs が有効、Auditd が有効)

実験 1, および 2 で補足するシステムコールは open, および openat とする。ただし、現時点では openat の結果から絶対パス名を求める部分の実装を完了していない。しかし、今回の実験はファイルアクセスによるオーバーヘッドを確認することが目的であり、正しいパスが得られなくとも性能評価の上で問題は無い。

次に、実験で用いた計算機の環境を以下に示す。

CPU
Intel Core i7 870 2.93GHz (4Core, Hyper Threading 有効)

RAM
4GB

OS
Ubuntu-14.04.1 (64bit)

Kernel
Linux-3.13.11 (Auditd の改変およびシステムコールの追加済み)

HDD
Hitachi HDP72050GLA360 (7200rpm, 16MB Cache, SATA II 3.0Gb/s)

6.3 実験結果

実験 1 の結果を表 3 に示す。世界の生成は 2.84 秒、削除は 0.71 秒、融合は 1.21 秒、融合内容表示は 0.05 秒でその操作が行えている。これらの実行時間は十分に実用性がある。

実験 2 の結果を表 4 に示す。条件 (2), (3), (4) は (1) と比較すると実行時間は 6%, 10%, および 14%増加している。この実行時間は補足するシステムコールの種類の増加に伴ってさらに増加すると予想される。そのため、捕捉や

表 3 基本操作の実行時間の結果

操作	実行時間 (秒)
生成	2.84s
削除	0.71s
融合	1.21s
融合内容表示	0.05s

その捕捉後の処理を高速化することが今度の課題としてあげられる。

7. 関連研究

トランザクション的な機能を OS に持たせる研究として、Spinellis[2] による研究が存在する。この研究では Mac OS X のソフトウェアのテスト環境を ZFS のスナップショット機能で生成している。また、fsevents を用いてテスト環境のファイルアクセスの捕捉と記録を行っている。また、Commit による融合の可否はファイルのタイムスタンプで判断される。そのため、本番環境のタイムスタンプの方が新しい場合、融合しようとすると衝突扱いになり、トランザクションの Abort でテスト環境自体が削除される。このとき、動作には問題がないにも関わらず、世界 OS で可能であった融合ができないという問題がある。世界 OS では融合が続行できる。さらに本研究での実装では wdiff, および wset コマンドによる環境の部分的な融合が行える。

また、Porter ら [9] も同じような機能を持つ OS の実装を行っている。本研究では、ユーザ空間からカーネル空間にわたって実装および拡張が行われているが、この研究ではカーネル空間の実装が主となっている。具体的には、Linux のシステムコールやデータ構造を拡張し、それらを利用するための API の追加が行われている。この拡張によって、トランザクションの性質をもつシステムソフトウェアの実装が容易になっている。論文では、その API を用いたトランザクションの性質を持った ext3 ファイルシステムの実装が行われている。Spinellis による実装と同様、Porter らの実装でも部分的な環境の融合が行えない。

本研究では Auditd によって、システムコール単位でファイルアクセスを補足している。捕捉の範囲を広げたものに、Dunlap ら [10] による研究がある。この研究では、User mode Linux[11] 上のゲスト OS における NIC やキーボードなどの周辺機器からの入力や割り込み全般の捕捉と記録を行っている。この記録を再生することで、ゲスト OS の状態を再現することが可能となっている。これによりゲスト OS のロールフォワードが実現でき、障害からの回復が行える。この研究と比較すると、本研究ではファイルを対象とした、迅速な回復が行える。

Docker[12] は Immutable Infrastructure の考え方に則っており、ホスト OS の設定によらずに、アプリケーションの動作環境を構築することが行える。さらに、設定の差分と環境をひとまとまりのイメージに固定し、配布することによって、環境を再利用することが可能となっている。Docker は世界 OS と同様、ファイルの書き込み差分を保持するために Aufs を利用し、アプリケーションが動作するコンテナに LXC を利用している。Docker と比較した本研究の特徴は、プロセスの移動、環境の多重継承、およびシステムコール捕捉を行うことである。これらの特徴により、本研究での世界 OS では複数環境を組み合わせたソフトウェアのテストを実施できるという利点が存在する。

表 4 パッチ当ての結果

条件区分	Aufs	Auditd	コマンドの実行時間 ((1) との増減率)
(1)	無効	無効	3.03s
(2)	無効	有効	3.20s(+6%)
(3)	有効	無効	3.32s(+10%)
(4)	有効	有効	3.46s(+14%)

8. おわりに

本研究では Linux における仮想化技術を用いて世界 OS の実装を行った。世界の実装には LXC を用い、ファイルのコピーオンライトには Aufs を用いる。また、融合に伴うプロセスの移動は Linux カーネルを改変して実装した。さらに、本研究では、世界ごとのファイルアクセスを補足している。この結果を用いることによってファイルアクセスの衝突の判定がより詳しく行える。この捕捉によって、融合で置き換えられるファイルから派生したファイルの存在を容易に知ることができる。さらに、世界の中身のディレクトリのファイルを融合前に編集することで、既存の世界 OS で行えなかった部分的な融合を実現している。

実装においては、Python によるユーザインタフェース、および世界管理プログラムを実装した。実験では基本操作や Linux カーネルソースのパッチ当てを行った。世界の基本操作はが数秒以内で完了させることができる。また、パッチ当ての実行時間は通常の LXC と比べ、親世界で 6%, ファイルの監視が無効の子世界で 10%, 子世界で 14%増加した。

今後の課題としては、補足できるシステムコールを増やすことがあげられる。また、世界の融合や捕捉後の処理の高速化、および WordPress といった高度なアプリケーションを動作させた場合の性能評価を行うことも課題としてあげられる。

参考文献

[1] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G.: The LOCUS distributed operating system, *ACM SIGOPS Operating Systems Review*, Vol. 17, No. 5, Acm, pp. 49–70 (1983).

[2] Spinellis, D.: User-level operating system transactions, *Software: Practice and Experience*, Vol. 39, No. 14, pp. 1215–1233 (2009).

[3] Sun, J., Shinjo, Y. and Itano, K.: The implementation of a distributed file system supporting the parallel world model, *The Third International Workshop on Advanced Parallel Processing Technologies*, Vol. 156, pp. 43–47 (1999).

[4] 石井孝衛，新城靖，板野肯三：プロセストレース機能を用いた世界 OS の実現，情報処理学会論文誌，Vol. 6, pp. 1702–1724 (2002).

[5] Linux Containers: https://lxccontainers.org. Accesed:

2015-09-25.

[6] aufs.sourceforge.net: http://aufs.sourceforge.net. Accessed: 2015-09-25.

[7] Audit: http://people.redhat.com/sgrubb/audit/. Accessed: 2015-09-25.

[8] ZFS on Linux: http://zfsonlinux.org/. Accessed: 2015-11-01.

[9] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A. and Witchel, E.: Operating System Transactions, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 161–176 (online), DOI: 10.1145/1629575.1629591 (2009).

[10] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A. and Chen, P. M.: ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay, *SIGOPS Oper. Syst. Rev.*, Vol. 36, No. SI, pp. 211–224 (online), DOI: 10.1145/844128.844148 (2002).

[11] User-mode Linux: http://user-mode-linux.sourceforge.net. Accessed: 2015-09-28.

[12] Docker - Build, Ship, and Run Any App, Anywhere: https://www.docker.com. Accessed: 2015-06-09.