Vol.2019-OS-145 No.13
2019/3/1

FreeBSDにおけるLinux互換コンテナを対象としたマイグレーション機構の実現

高川 雄平[1,a)]　松原 克弥[1,b)]

概要：Linux において確立されたコンテナ型仮想化技術は，Docker の登場と普及にともなって，標準化団体 Open Container Intiative(OCI) 主導でコンテナ仕様が定義され，現在では，Windows や macOS などの他の OS 環境でもコンテナを利用できるようになった．FreeBSD 環境で動作する Docker では，FreeBSD が持つ Linux バイナリ互換機能などを活用してコンテナを実現している．しかし，FreeBSD 上の既存コンテナ実装は，プロセスのアイソレーション機能やマイグレーション機能など，OCI コンテナ仕様と比較して，その実現が十分とはいえない箇所が存在する．本研究では，FreeBSD 上で OCI 仕様に準拠するコンテナを実装し，さらに，そのコンテナのマイグレーションを実現する手法について提案する．

1. はじめに

近年，クラウドコンピューティング環境において，Docker[6] や LXC[9]，FreeBSD Jail[5] といったコンテナ型仮想化システムの導入が拡がっている．コンテナ型仮想化とは，OS 上の計算資源や名前空間を隔離・制限することによって，特定のプロセス群に対して，ホストと異なる実行環境（以降，コンテナ）を提供する技術である．ハードウェア環境を仮想化する従来技術と比較して，複数の異なる OS を共存させせることはできないが，仮想環境であるコンテナの生成が軽量で，実行時のオーバヘッドも少ない．コンテナ型仮想化システムの中でも特に広く利用されている Docker は，現在，標準化団体 Open Container Initiative(OCI) が提唱するコンテナ仕様の標準規格に沿って開発が進められており，複数のコンテナ関連ツールやサブシステムとの柔軟な連携が可能となっている．例えば，CRIU(Checkpoint/Restore in Userspace)[13] を連携させることで，動作中のコンテナを別のマシンに移動させる，コンテナマイグレーションを実現できる．

コンテナ型仮想化は Linux において確立された技術ではあるが，用途に応じて他の OS が適している場合もあり，複数の OS を比較評価する試みも行われている [7][8]．実際，サーバ OS のシェアは，Linux が 36%で最も多いが，Windows が 31%，BSD 系 UNIX も 6.2%のサーバ環境で利用されている [14]．また，動画配信サービスの Netflix では，動画データ配信を担う CDN (Content Delivery Network) システムの OS プラットフォームとして，FreeBSD を積極的に採用している事例も見られる [10]．

本研究では，複数の異なる OS を状況に応じて動的に切り替えることで，アプリケーション／サービスの性能や可用性を向上させることを目的として，異種 OS 間のコンテナマイグレーション機構の実現を目指している．本稿では，Linux と FreeBSD 間のコンテナマイグレーション実現の要として，FreeBSD 上で動作する OCI 仕様準拠のコンテナ型仮想化システム runC[1] とマイグレーションツール CRIU の実現手法について述べる．

以降，本稿では，第 2 章で FreeBSD 上で実現されている既存コンテナ実装，第 3 章で OCI コンテナランタイム仕様について紹介する．その後，第 4 章ではコンテナランタイム runC，第 5 章ではマイグレーションツール CRIU を FreeBSD へ移植する手法について述べる．第 6 章では，runC および CRIU の実現手法として提案するコンテナ型仮想化実現技術，および，マイグレーション技術を評価するために行った実験結果について示す．最後に，第 7 章でまとめと今後の課題について述べる．

2. FreeBSDにおける既存コンテナ実装

本研究では，FreeBSD 上で動作するコンテナランタイム runC とマイグレーションツール CRIU を実現する．本実現の有用性を示すために，FreeBSD で利用できる既存コンテナ実装システムについて紹介する．

[1] 公立はこだて未来大学
Future University Hakodate
a) g2118021@fun.ac.jp
b) matsu@fun.ac.jp

© 2019 Information Processing Society of Japan

2.1 Docker

公式リポジトリ FreeBSD Ports には，freebsd-docker[11] と名づけられた FreeBSD 移植版 Docker 実装が存在する．freebsd-docker は，FreeBSD のコンテナ技術である Jail と ZFS ファイルシステムの機能を利用して，コンテナを実現している．しかし，2015 年以降い更新が行われておらず，当時のバージョンである Docker v1.9 の仕様に沿って実装されている．また，DockerHub からダウンロードできるイメージの多くが Linux ELF 形式であるため，Linux 互換機能である Linuxulator の機能に実装も依存している．例えば，freebsd-docker のネットワークプロキシである docker proxy が Linux プロセスとして実現されている．freebsd-docker は，OCI 仕様にも準拠していないため，内部モジュールが機能分割されていない．そのため，containerd などの上位ランタイムや CRIU などの関連ツールとの連携も困難である．

2.2 FreeBSD VPS

FreeBSD Virtual Private System(VPS)[12] は，FreeBSD 上で実現したした独自仕様のコンテナ（以降，VPS インスタンス）とその VPS インスタンスのマイグレーション機能を実現している．FreeBSD Jail のカーネル実装では，jail システムコールを介した jail 構造体に対する制御によりコンテナの操作を可能にしているが，コンテナの実体はカーネル内の prison 構造体で管理されている．一方，FreeBSD VPS では，jail 構造体に相当する構造体がいくつか用意されている．また，prison 構造体に当たる vps 構造体には prison 構造体が含まれており，FreeBSD Jail を拡張したものとなっている．また，Jail には実装されてない，ネットワークスタックの分離やプロセステーブルの分離も実現している．FreeBSD VPS におけるマイグレーションでは，カーネルが保持する Process Control Block(PCB) やカーネルスタックなどを利用している．

FreeBSD VPS は，VPS インスタンスやマイグレーションの実装にともなって，カーネルコードの多くの箇所に変更を加えているため，利用するためにカーネルの再ビルドを必要とする．

3. OCI コンテナランタイム仕様

現在の Docker*1 が準拠している OCI コンテナランタイム仕様について述べる．OCI が定めるランタイムの仕様は，図 1 のような構造で定義される．OCI 仕様では，コンテナランタイムを上位コンテナランタイムと下位コンテナランタイムに分離している．上位コンテナランタイムは，Docker や Kubernetes からの命令を Container Runtime Interface(CRI) 仕様を介して受け取り，下位コンテナランタイムがコンテナを作成するための設定ファイルを作成する．

*1 本稿執筆時点の最新バージョンは，18.09 である．

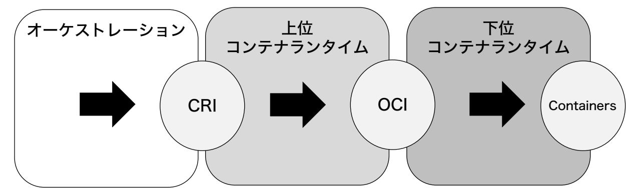

図 1 コンテナランタイム構成とインタフェース仕様 [4]

代表的な上位ランタイムとして containerd[2] が存在する．containerd は，containerd-shim という抽象化レイヤで，下位コンテナランタイムを切り替えている．下位コンテナランタイムは，実際にコンテナを作成する役割を持ち，リソースを制限やリソースの隔離の設定をする．代表的な下位コンテナランタイムとして，Linux には runC，Windows には runhcs[3] が存在する．

OCI のコンテナ仕様は Linux のコンテナ作成機能である Namespace や cgroups に基づいて決められている．Namespace では以下の 6 つのリソースを隔離している．

- IPC 名前空間
  プロセス間通信で利用される System V IPC オブジェクトや POSIX メッセージキューを分離する．
- Mount 名前空間
  ファイルシステムのマウントポイントを分離する．同じファイルパスに対して，コンテナごとに異なるファイル実体を対応づけることが可能である．
- UTS 名前空間
  ホスト名と NIS ドメイン名を分離する．Network 名前空間と組み合わせることで，ドメイン名をコンテナごとに変更可能である．
- Network 名前空間
  ネットワークに関するリソースであるネットワークデバイス，プロトコルスタック，IP ルーティングテーブルなどを分離する．異なるコンテナとは veth を利用することで通信可能である．
- PID 名前空間
  プロセス番号を分離し，異なるコンテナであれば同じ番号の存在を許可する．特別な意味をもつ PID 1 として実行される init プロセスも複数のコンテナで実行可能である．
- User 名前空間
  UID(User ID) や GID(Group ID)，root ディレクトリを分離し，UID に対して付与する権限や属性をコンテナごとに変更可能である．
- cgroups 名前空間

© 2019 Information Processing Society of Japan

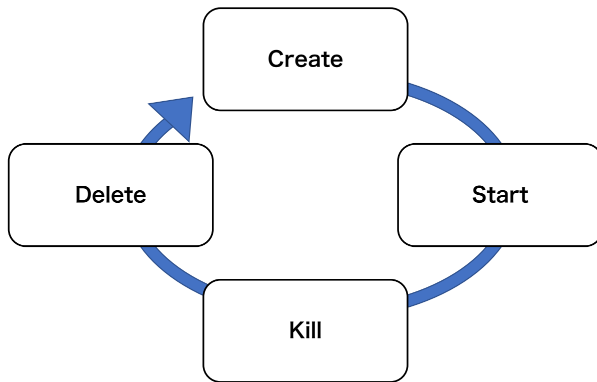

図 2　OCI 仕様のコンテナライフサイクル

自コンテナに関する cgroups の設定以外を隠蔽する．

cgroups では数多くのリソースを制限することができるが，本稿では Kubernetes が制限するリソースを対象とし，以下に 6 つのリソースを示す．

- memory
  メモリの使用量をバイト単位で制限可能である．
- cpushare
  CPU 使用率を他のコンテナとの相対比率で制限可能である．
- cpuquota, cpuperiod
  CPU の使用率を period と quota の時間比で設定可能である．
- hugepage
  HugeTLB と呼ばれる仮想メモリのページサイズを 4KB 以上割り当てる技術の割り当て上限を設定可能である．
- devices
  特定のデバイスに対するアクセス制御が可能である．
- cpuset
  使用する CPU コアを指定可能である．

OCI 仕様のコンテナのライフサイクルは，図 2 に示したように，作成，起動，停止，削除という流れになるため，一度停止したコンテナを再度起動することはできない仕様である．

3.1　runC

runC は，OCI 仕様に準拠した下位コンテナランタイム実装である．コンテナの作成，起動，停止，削除を実現するだけでなく，CRIU と連携したライブマイグレーションにも対応している．

runC のコンテナ作成では，chroot を用いてファイルシステムを分離する rootfs とコンテナの設定を記述した config.json を用意する．この rootfs と config.json を合わせて runC Bundle と呼ぶ．rootfs には，ルートディレクトリにあるような var ディレクトリ，bin ディレクトリ，共有ライブラリなどを展開する必要がある．config.json にはコンテナの作成に必要な記述があり，コンテナ内で実行するコマンドのパスや引数，環境変数，Namespace によるリソースの分離や cgroups によるリソースの制限，ケーパビリティなどの設定する．コンテナの作成では，config.json に記述されたデータを全て適用したインスタンスを作成する．コンテナの起動では，コンテナ ID を割り振り，インスタンスの起動，つまり，プロセスの実行と Namespace や cgroups への設定の反映を行う．コンテナの停止では，コンテナのプロセスにシグナルを送信する．コンテナの削除では，コンテナ ID を削除し，インスタンスを削除する．

表 1　Linux Namespace と FreeBSD Jail との対応

隔離機能	Linux	FreeBSD
IPC	Namespace	Jail
Mount		
UTS		
Network		
cgroups		
PID		△ Jail
User		

4.　FreeBSD 向け runC の改良

本研究では，Linux と FreeBSD 間のコンテナマイグレーションを実現することを目的として，FreeBSD 上で動作する OCI 準拠コンテナランタイムを実現する．実装対象として，Hongjaing Zhang[15] が 2015 年に FreeBSD 向けに移植を行った runC 実装をベースとする．本 runC 実装は，runC の持つコンテナ実現機能のうち，リソース隔離のみを FreeBSD 上で実現している．Linux 上で動作するコンテナを FreeBSD 上へマイグレーションして実行を継続するためには，リソースの制限機能を追加実装する必要がある．OCI 仕様におけるコンテナ設定項目は，Linux のリソース隔離・制限機能をもとにして定義されている．実際，runC は，Linux のリソース隔離・制限機能をベースとして実装されているため，本実現では，Linux に依存する機能実装を FreeBSD の機能による実装に置き換える．Linux から FreeBSD へ機能を置き換えるためには，Linux の機能と FreeBSD の機能を対応づけを行う必要がある．以下に，リソース隔離機能とリソース制限機能のそれぞれについて，Linux と FreeBSD の各機能の対応づけを示す．

4.1　リソース隔離機能

Linux の隔離機能である Namespace との対応づけを表 1 に示す．FreeBSD ではコンテナ型仮想化を FreeBSD Jail で実現している．FreeBSD Jail では Linux Namespace と同様の名前空間の隔離ができるが，PID 名前空間に関して

© 2019 Information Processing Society of Japan

表 2　Linux cgroups と FreeBSD の既存機能との対応

機能概要	Linux cgroups	FreeBSD の既存機能
メモリ使用量	memory	RCTL memoryuse
相対的 CPU 使用率	cpushare	×
CPU 使用率	cpuquota,period	◇ RCTL の pcpu
ページサイズ	hugepage	△ superpage
デバイスアクセス	devices	devfs
CPU コアの指定	cpuset	cpuset

は多重化ができず，User 名前空間に関しては分離されていない．FreeBSD では，名前空間を隔離するためには，jail コマンドを実行し，その子プロセスとして対象のプロセスを実行する方法と jail システムコールや jail_set システムコールを直接対象プロセスで実行する方法がある．既存の FreeBSD 向け runC では，jail コマンドを実行し，子プロセスとして実行する方法を取る．

4.2　リソース制限機能

Linux の制限機能である cgroups との対応づけを行う．対応づけを表 2 に示す．なお，今回対象とする計算資源の制限対象は，Kubernetes がサポートしている計算資源とする．CPU 使用率は，時間で設定する cgroups の cpuquota と百分率で設定する RCTL の pcpu に対して，変換を行うことで対応づけをすることができる．cpuperiod は CPU の再割り当てまでの時間であり，その割り当ての中で cpuquota 分だけ対象のプロセスに CPU を割り当てる．式 (1) では，CPU 使用率を百分率で制限する FreeBSD RCTL の pcpu と時間の比率で制限する Linux cgroups の cpuquota, cpuperiod の変換式である．

$$pcpu = \frac{cpuquota}{cpuperiod} \times 100 \quad (1)$$

FreeBSD では，相対的 CPU 使用率とページサイズについては扱わないこととする．FreeBSD のスケジューラの nice 値の扱いが異なるため，相対的 CPU 使用率の制限ができない．Hugepage のページサイズに関しては，システム全体に対して制限することしかできず，コンテナごとに設定することはできないため，HugeTLB を実現する FreeBSD の機能 superpage の機能拡張が必要になる．

Linux では制限値を超えた場合，制限値を超えないように割り当ての拒否や速度の制限が行われるが，RCTL ではどのような操作をするか指定することができる．本実装では，制限値を超えたリソースの割り当てを許可しない”deny”を設定する．

Linux でデバイスを制限する場合，デバイスをメジャー/マイナー番号で指定し，書き込み，読み込み，特殊ファイル作成をそれぞれ許可するかを config.json に設定を記述する．一部のデバイスは，デフォルト設定として runC の実装にハードコードされている．一方，FreeBSD では，デバイス制限をメジャー/マイナー番号ではなく，デバイス名で指定し，書き込みと読み込みの両方を同時に許可するかしないかを決める．Linux と FreeBSD では，同じデバイス名を指していても，メジャー/マイナー番号が異なるため，Linux 向けの設定が記述された config.json からでは，デバイス名を特定することはできない．本実装では，Linux のデフォルトとして runC の実装にハードコードされたデバイスのみを，FreeBSD で制限するようにし，メジャー/マイナー番号からデバイス名を特定できるようにする．

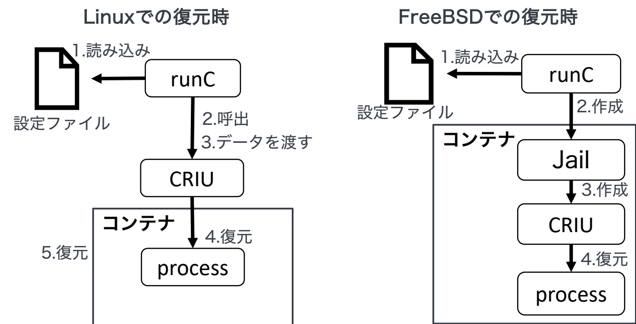

図 3　コンテナ復元の概要 [4]

4.3　マイグレーション機能

図 3 では，Linux の既存のコンテナ復元機能と FreeBSD に追加するコンテナ復元機能の概要を示している．Linux では，コンテナの復元時に runC が設定ファイルである config.json から前述の設定を読み込み，CRIU に Google Protocol Buffer の形式で Unix Domain Socket を介してデータを渡し，CRIU がプロセスの復元とコンテナの復元を行う．

本稿のコンテナマイグレーションでは，runC が持つコンテナ作成機能と設定ファイル config.json を再利用する．runC が config.json を読み込んだ後に，Jail コマンドでコンテナを作成し，そのコンテナ内で CRIU がプロセスを復元する．CRIU はプロセスのみに関して操作を行い，runC はコンテナのみに関して操作するような構成となる．コンテナの情報に関しては，設定ファイルをそのまま利用するため，取得する必要がない．ただし，runC を介入しないで変更された制限値は config.json には反映されないため，取得方法を検討する必要がある．

次の章では，FreeBSD のプロセスマイグレーションを実現するための CRIU の実装について説明する．

5.　FreeBSD 向け CRIU の実現

CRIU は，Linux 上で動作するプロセスの状態を取得す

© 2019 Information Processing Society of Japan

る機能，および，取得した状態を新たなプロセス内で復元する機能を実現することで，プロセスをマイグレーションすることを可能にしている．本実現では，FreeBSD の Linux emulation (Linuxulator) を使って動作する Linux 互換プロセスを対象として，プロセス状態の取得と復元の機能を CRIU に実装する．プロセスの状態には，CPU レジスタやメモリのようなプロセスを構成する基本的要素の他に，プロセスが開いたファイルや TCP コネクションの状態などが存在する．本章では，対象プロセスの実行状態，ファイルアクセス状態，ネットワーク通信状態のそれぞれの取得と復元を実現する方法について述べる．

5.1 プロセス実行状態

プロセスを別のマシンに移動して，続きから動作させるには，CPU レジスタの情報とメモリの情報を取得し，書き込むことで復元する必要がある．

メモリの情報は procfs の mem ファイルから read システムコールと write システムコールを用いることで，読み込みと書き込みが行える．メモリに関しては，メモリレイアウトを考慮する必要がある．メモリレイアウトとは，仮想メモリ空間におけるデータ領域やスタック領域などの配置のことである．メモリの情報を書き込む前に，mmap システムコールと munmap システムコールを使ってメモリレイアウトを変更する必要がある．

CPU レジスタの情報は ptrace システムコールを用いることで，取得と書き込みを行うことができる．

Linuxulator を用いた Linux 互換プロセスの復元では，その実行タイミングに特別な対処を必要とする．通常，プロセスを復元する場合は，ptrace システムコールの TRACEME を有効にして，exec 系システムコールで対象の実行ファイルを実行する．その後，親プロセスが対象プロセスのエントリポイントでフックした時に，レジスタやメモリの情報を復元する．しかし，Linux 互換プロセスでは，Linux 互換機能のための初期化処理が必要であるため，プロセスのエントリポイントは復元処理実行の適切なタイミングではない．Linux 互換初期化処理が完了しているタイミングの取得方法として，ブレイクポイントである int3 命令を main 関数の開始命令に書き込む．int3 命令により main 関数実行直前のタイミングで制御を取得し，プロセス状態を示すメモリとレジスタの値を復元する．

5.2 ファイルアクセス状態

マイグレーションの対象であるプロセスがファイルへアクセスしている場合，OS カーネル内で，ファイルパス，ファイルディスクリプタ番号，オフセット，アクセスモードやフラグなどファイルに関する情報がプロセスと関連づけて保持されている．Linux では procfs の fdinfo ディレクトリから全ての情報を取得できるが，FreeBSD には fdinfo ディレクトリが存在しない．代わりに devfs の fd ディレクトリを介して当該情報をカーネル内から取得することもできるが，fd ディレクトリへのアクセスは自プロセスしか許可されず，他プロセスの情報を取得するためのアクセスはできない．そこで，FreeBSD で他プロセスに関する情報を取得できる __sysctl システムコールを利用して，他プロセスが持つファイル情報を取得できる libprocstat ライブラリを利用する．

ファイル情報の復元は以下の手順で行う．

(1) 取得したファイルパスのファイルを open システムコールを用いて開く
(2) open システムコールのオプションには，取得したアクセスモードやフラグを設定する
(3) dup2 システムコールを用いて，ファイルディスクリプタの番号を変更する
(4) lseek システムコールを用いて，ファイルオフセットの位置を変更する
(5) exec 系システムコールで対象の実行ファイルを実行する

5.3 ネットワーク通信状態

TCP コネクション状態を取得し復元することを TCP repair と呼ぶ．TCP コネクションの状態とは，送信キューのデータ，受信キューのデータやシーケンス番号などを指す．TCP repair は Linux Kernel 3.6 に組み込まれた機能である．本実装では，Linux の TCP repair を FreeBSD に移植するために，FreeBSD カーネルのネットワークに関する処理に手を加える．

TCP repair を行うには，以下の要件を満たす必要がある．

- 送受信キューのデータを取得
- 送受信キューにデータを反映
- シーケンス番号の取得と反映
- FIN パケットを送信せずにソケットを閉鎖
- SYN パケットを送信せずにソケットを接続

図 4 に示す番号はデータの状態を示しており，それぞれ以下の状態を示している．

(1) ユーザ空間からカーネル空間にコピーされていないデータ
(2) カーネル空間にあるが，まだネットワークに送信されていないデータ
(3) ネットワークに送信されたが，まだ受信されてないデータ
(4) 受信してカーネル空間にあるが，まだユーザ空間に読まれていないデータ
(5) ユーザ空間のプロセスが受け取ったデータ

(1) と (5) のデータはユーザ空間で取得し，復元することができるデータである．(3) のデータはパケットロスした際に，TCP の再送制御によって再送されるデータである．

© 2019 Information Processing Society of Japan

(2) と (4) のデータはユーザ空間からは取得することのできないデータである．

Linux では，getsockopt/setsockopt システムコールのオプションに，TCP_REPAIR，TCP_REPAIR_QUEUE，TCP_QUEUE_SEQ を追加している．TCP_REPAIR オプションを有効にした場合，SYN パケットを送信しない connect システムコールと FIN パケットを送信しない close システムコールを実行することができる．TCP_REPAIR_QUEUE オプションでは，取得または復元する対象のキューを送信キューと受信キューから選択する．TCP_REPAIR_SEQ オプションでは，シーケンス番号の取得と復元を行う．

FreeBSD カーネルでは，sendmsg システムコールを使った場合，送信キューにデータがコピーされる．recvmsg システムコールを使った場合，受信キューからデータがコピーされる．既存のカーネル機能では，送信キューからデータをコピーする手段と受信キューにデータをコピーする手段がない．そのため，setsockopt システムコールの TCP_REPAIR_QUEUE を使うことで，recvmsg システムコールを用いて，指定したキューからデータを取得し，sendmsg システムコールを用いて，指定したキューにデータをコピーするようにカーネルを変更する．

FreeBSD カーネルでは，シーケンス番号を tcpcb 構造体で管理している．送信キューのシーケンス番号は，メンバ変数 snd_nxt，受信キューのシーケンス番号はメンバ変数 rcv_nxt に格納されている．getsockopt/setsockopt システムコールの TCP_REPAIR_SEQ を用いて，指定したキューのシーケンス番号を取得，反映できるようにカーネルを変更する．

また，Linux と FreeBSD とでは，ネットワークスタックの実装が異なるため，実装を移植したとしても，ウィンドウサイズが更新されず，通信速度が極端に低下するという問題がある．FreeBSD 上でウィンドウサイズが更新されないのは，フロー制御のウィンドウサイズを更新時に比較される送信キューのウィンドウの開始位置である SND.WL2 が取得されず，復元できないためである．FreeBSD におけるウィンドウサイズ更新の評価に関するフローチャートを図 5 に示す．SND.WL2 は利用できないため，条件式 (3)，(4) は偽となり，更新されない．そのため，条件式 (2) になる場合が更新されない原因となる．本稿のアプローチとしては，復元後最初の評価のみを必ず真にするように，条件式 (1) 満たすようにする．つまり，SND.WL1 をシーケンス番号よりも小さく保つように復元する．SND.WL1 と SND.WL2 は評価直後に正しい値が設定されるため，問題がない．

6. 実験

本稿で提案する FreeBSD 向け OCI 準拠コンテナランタイムの実現とコンテナマイグレーション実現手法の有効性を評価するために，提案したプロセスの実行状態と TCP コネクションのマイグレーションのオーバヘッド，および，リソース制限機能を実装した runC におけるコンテナの起動と復元にかかるオーバヘッドを計測した．

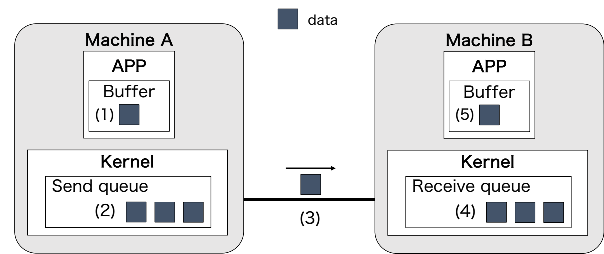

図 4　TCP repair の手順

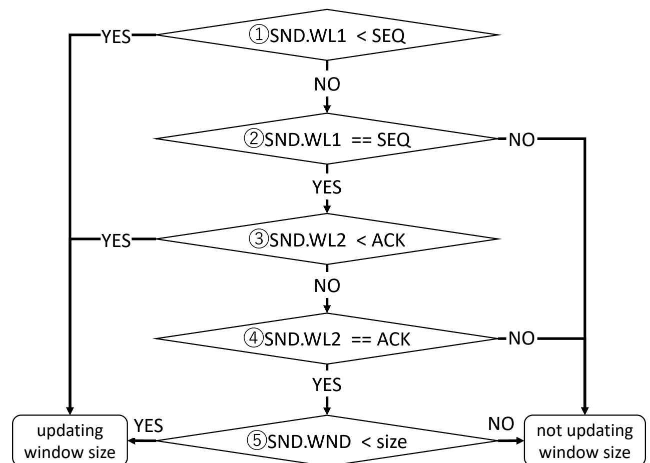

図 5　ウィンドウサイズ更新のフローチャート

表 3　プロセスマイグレーションに用いた実験環境

OS	Ubuntu 16.04 LTS (Linux kernel: 4.8.0-36-generic)
	FreeBSD 11.0-RELEASE
CPU	Intel(R)Core(TM)i3 2.4GHz
RAM	4GB
HDD	MQ01ABD050
Network	1Gbps Ethernet

6.1　プロセス実行状態のマイグレーション

CRIU と実装した技術を用いて Linux と FreeBSD のプロセス実行状態のみに関して，保存と復元にかかる時間を計測した．プロセスマイグレーションの実験環境を表 3 に示す．図 6 は 1000 回の結果の平均値を示している．

復元にかかる時間はどのパターンでも 1.00 ms 前後であった．しかし，保存にかかった時間は，Linux では 70.16 ms だったのに対し，FreeBSD では Linux の 2 倍である 142.64 ms かかった．

メモリ領域のサイズ取得には精度の違いがあり，取得するメモリサイズが異なっていることが原因である．Linux

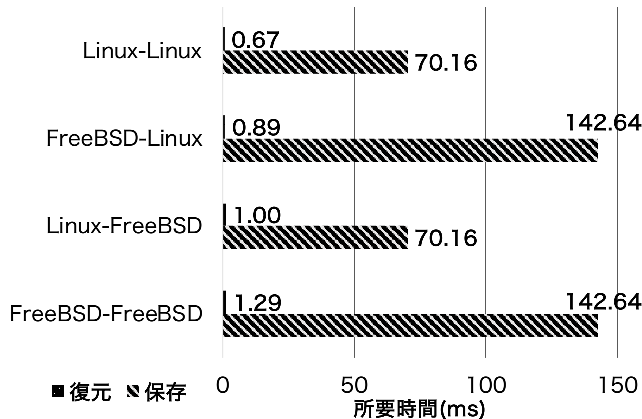

図 6　プロセス実行状態のみの異種 OS 間マイグレーションに要した時間

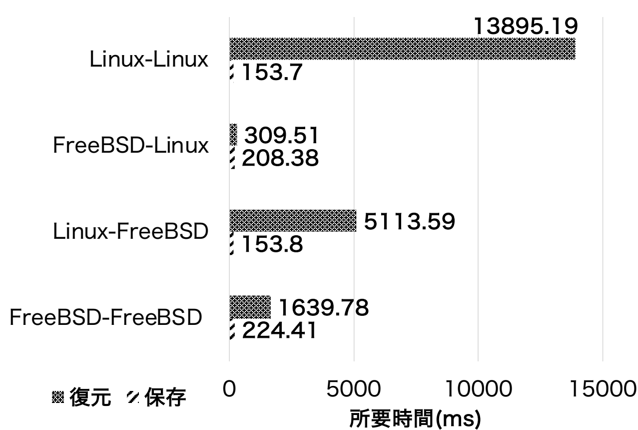

図 7　TCP コネクションをもつプロセスの異種 OS 間マイグレーションに要した時間

ではデータ領域とヒープ領域は異なる領域として認識されるが，FreeBSD では同じ領域として認識される．そのため，ヒープ領域を利用していない今回のテストプログラムが対象だと Linux と FreeBSD で大きな差が出ると考えられる．

6.2 TCP コネクションのマイグレーション

前項同様の実験環境を用いる．Linux と FreeBSD の TCP コネクションを保持するプロセスに関して，保存と復元にかかる時間をそれぞれ 50 回計測した．図 7 は 50 回の結果の平均値を示している．

保存にかかる時間は Linux では 153.8 ms 前後であるのに対し，FreeBSD では 216.39 ms かかった．Linux で保存したデータを Linux で復元する際には 13,895.19 ms かかり，Linux で保存したデータを FreeBSD で復元する際には 5,113.59 ms かかった．FreeBSD で保存したデータを Linux で復元する際には 309.51 ms かかり，FreeBSD で保存したデータを FreeBSD で復元する際には 1,639.78 ms かかった．

表 4　runC の計測に用いた実験環境

OS	FreeBSD 11.2-RELEASE
CPU	Intel(R)Core(TM)i5-7200 3.40GHz
RAM	8GB
HDD	1TB

Linux で保存したキューの平均サイズは 121,595 B であるのに対し，FreeBSD で保存したキューの平均サイズは，Linux の約 1/4 となる 33,164 B であった．FreeBSD で保存したデータを復元するのにかかった時間よりが Linux で保存したデータを復元するのにかかった時間よりかなり少ないのは，復元すべきキューのサイズが異なることが原因である．

TCP repair 前のパケット到着時間から TCP repair 後のパケット到着時間の差は，Linux が 285.02 ms なのに対し，FreeBSD は 1,926.64 ms であった．FreeBSD で取得したデータを FreeBSD で復元する時間が，Linux で復元するよりも多くかかったのは，今回実装した FreeBSD の TCP repair にかかる時間が多いことが原因である．

6.3 コンテナの起動と復元

runC の拡張によるオーバーヘッドの計測に用いた実験環境を表 4 に示す．基準値となる拡張前の runC でコンテナの起動に要した時間を 50 回計測した．また，リソース制限機能，コンテナ復元機能を持つ runC でコンテナの起動に要した時間とコンテナの復元に要した時間をそれぞれ 50 回計測した．実験の対象となるコンテナ上で動作するプロセスは，ファイルの状態を持っているが，TCP コネクションの状態を持っていないプロセスである．

図 8 は 50 回の結果の平均値を示している．”STANDARD”は基準値であり，”START”は拡張した runC のコンテナ起動に要した時間，”RESTORE”は拡張した runC のコンテナ復元に要した時間である．”create”はコンテナの作成に要した時間，”start”は対象プロセスが実行開始するまでの時間，”restore”は対象プロセスが CRIU によって復元されるまでに要した時間である．本稿の実装に関係する部分は，コンテナの作成に要した時間で，”START”はリソース制限機能を追加したオーバーヘッド，”RESTORE”はリソース制限機能とコンテナ復元機能を追加したオーバーヘッドである．

リソース制限機能を追加したオーバーヘッドは，基準値と比べて 1.46 ms 増加しているが，基準値の全工程に要する時間の 1.23%だった．リソース制限機能とコンテナ復元機能によるオーバーヘッドは，基準値と比べて 2.29 ms 増加しているが，基準値の全工程に要する時間の 1.93%だった．また，対象プロセスが実行開始するまでの時間の標準偏差は，リソース制限機能の場合は 2.62 ms，リソース制

© 2019 Information Processing Society of Japan

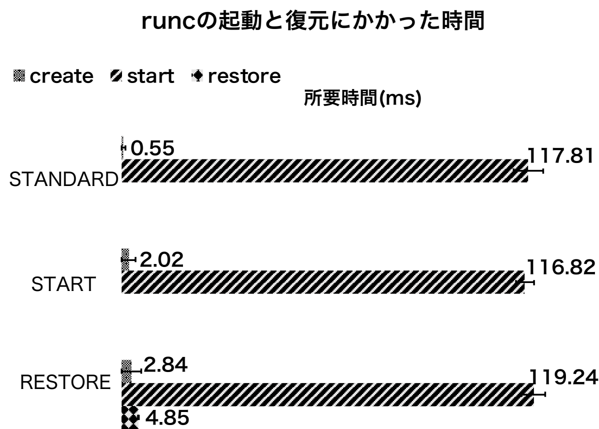

図 8 runC のコンテナ起動とコンテナ復元に要した時間

限機能とコンテナ復元機能の場合は 3.43 ms となっている．つまり，もっとも時間のかかる工程のブレよりもオーバヘッドは小さく，小さいオーバーヘッドで拡張することができたと言える．

7. おわりに

本稿では，OCI の標準規格に基づいたコンテナ型仮想化を FreeBSD で実現するために，Linux 向けに開発されたコンテナランタイム runC とマイグレーションツール CRIU の移植手法を提案した．現行の FreeBSD 向け runC 実装において実現されていないリソース制限機能を FreeBSD 上で実現するために，Linux と FreeBSD のそれぞれが持つリソース制限機能を比較し，その対応づけを行い，FreeBSD 向け runC 現実装へリソース制限機能を実装した．その際，相対的 CPU 利用率による制限や HugeTLB におけるページサイズ制限など，現行の FreeBSD が提供する機能だけでは実現できないものがあることが判明した．また，CPU 利用率による制限に指定できる値の単位が異なるために，OCI/Linux 仕様の設定値を FreeBSD 仕様の値に変換する式を定義した．デバイスファイルのメジャー・マイナー番号で指定されるデバイス制限機能では，Linux と FreeBSD 間でその番号割り当てが異なるため，/dev/null など両 OS での存在が自明なデバイスのみメジャー・マイナー番号の変換を行うこととした．

また，FreeBSD 上でのコンテナマイグレーションを実現するために，CRIU を FreeBSD へ移植した．Linuxulator 上で動作するプロセスの復元では，main 関数の開始命令を int3 命令（ブレイクポイント）を置き換えることで，Linux 互換機能の初期化完了後のタイミングで制御を捉えて，プロセス状態の復元処理を実行する．ファイルアクセス状態のマイグレーションでは，libprocstat ライブラリを用いて対象プロセス内のファイルアクセス状態を取得し，open，dup2，lseek 等のシステムコールを駆使して各ファイルアクセス状態を復元する．ネットワーク通信状態の取得と復元では，TCP コネクションの状態を示すカーネル内送受信キュー内のデータとシーケンス番号の取得，マイグレーション先での送受信キューへのデータの復元，ウインドウサイズなどのフロー制御を CRIU から行えるように FreeBSD カーネル内のネットワークプロトコル・スタックを改造した．

今後の課題は，本実現で対応付けができなかったリソース制限機能への対処がある．また，FreeBSD 上のコンテナ生成後に動的に変更されたリソース制限設定の取得と復元機能を確立することで，CRIU におけるコンテナマイグレーション機能の実現を行う．これらの課題に対処することで，本実現のコンテナランタイムを用いた異種 OS 間コンテナマイグレーションの実現を目指したい．

参考文献

[1] Ben Golub, S. H.: Day 1 Keynote (2015). DockerCon 15.
[2] containerd authors, T.: containerd, (online), available from ⟨https://containerd.io/⟩ (accessed 2019-02-04).
[3] Cooley, S.: Container platform tools on Windows, (online), available from ⟨https://docs.microsoft.com/virtualization/windowscontainers/deploy-containers/containerd⟩ (accessed 2019-02-04).
[4] Hasegawa, M.: runc (2018). JapanContainerDays v18.04.
[5] henning Kamp, P. and Watson, R. N. M.: Jails: Confining the omnipotent root, *In Proc. 2nd Intl. SANE Conference* (2000).
[6] Inc., D.: The Docker Containerization Platform., (online), available from ⟨https://www.docker.com/⟩ (accessed 2019-01-24).
[7] Larabel, M.: FreeBSD 12.0 vs. DragonFlyBSD 5.4 vs. TrueOS 18.12 vs. Linux On A Tyan EPYC Server.
[8] Larabel, M.: Windows Server 2019 vs. Linux vs. FreeBSD Gigabit & 10GbE Networking Performance, (online), available from ⟨https://www.phoronix.com/scan.php?page=article&item=windows-linux-10gbe⟩ (accessed 2019-02-04).
[9] LinuxContainers.org: LinuxContainers.org., (online), available from ⟨https://linuxcontainers.org⟩ (accessed 2019-01-30).
[10] Looney, J.: Netflix and FreeBSD (2019). FOSDEM 2019.
[11] MateuszPiotrowski: Docker on FreeBSD, (online), available from ⟨https://wiki.freebsd.org/Docker⟩ (accessed 2019-01-30).
[12] Ohrhallinger, K. P.: Virtual Private System for FreeBSD, *EuroBSDCon 2010* (2010).
[13] Project, C.: CRIU Main page, (online), available from ⟨https://criu.org/⟩ (accessed 2019-01-30).
[14] W3Techs: Usage of operating systems for websites, (online), available from ⟨https://w3techs.com/technologies/overview/operating_system/all⟩ (accessed 2019-02-04).
[15] Zhang, H.: Implement FreeBSD runc with the help of Jail, (online), available from ⟨https://lists.freebsd.org/pipermail/freebsd-jail/2017-July/003400.html⟩ (accessed 2019-01-30).

© 2019 Information Processing Society of Japan