動的OS切り替えを目的としたFreeBSDにおけるLinux互換プロセスのマイグレーション機構

松原 克弥　髙川 雄平

これまでのコンピュータ技術進歩の過程において，基盤ソフトウェアであるオペレーティングシステム（OS）の研究開発が盛んに行われ，数多くの OS 実装が利用可能となった．デスクトップ PC やスマートフォン，エンタープライズサーバなどの一部のプラットフォーム向けでは，Windows や Linux などの少数の OS が寡占状態にあるが，すべてのシステムにおいて特定の OS が独占して採用されることはなく，現在でも，システムの目的や用途，プラットフォームの特性などに応じて複数の OS を使い分けている．著者らは，特性の異なる複数の OS を状況の変化に応じて動的に切り替えることで，OS 上で稼働するサービスの負荷耐性や性能の改善を試みる手法を提案している．本論文では，Linux と FreeBSD を対象として，サービス稼働中にそのシステム基盤である OS の動的な切り替えを可能とすることを目的として，FreeBSD における Linux 互換プロセスのマイグレーション機構の実現手法についてまとめる．Linux におけるプロセスマイグレーション機構の標準ツールである CRIU とのチェックポイントデータの相互互換性を維持することで，Linux と FreeBSD 間で同一のプロセスをマイグレーションする仕組みを実現する．本実現法では，FreeBSD カーネルと Linux カーネルの OS 機能の違いだけでなく，内部構造やインタフェース仕様の違いにも着目し，CRIU の各機能と同等の機能を FreeBSD 上で実現する際の課題や実装の詳細について解説する．

In the process of technological progress of the computer, research and development of the operating system (OS), which is the primary software, has been actively carried out, and several OS implementations have become available in use. Although a few operating systems, such as Windows and Linux, are in an oligopoly for some platforms, such as desktop PCs, smartphones, and enterprise servers, no one OS has a monopoly for all systems. Even today, each of the existing multiple OSes is selectively used in different ways, depending on the purpose and platform characteristics of the target system. The authors propose a method to improve the load tolerance and performance of services running on an OS by dynamically switching between multiple OSes with different characteristics according to the changing situation. In this paper, we summarize the implementation of a Linux-compatible migration mechanism for Linux and FreeBSD to enable the dynamic switching of the operating system on the system's infrastructure during the service operation. By maintaining the mutual compatibility of checkpoint data with CRIU, the standard tool for process migration mechanism in Linux, we implement a mechanism for migration of the same process between Linux and FreeBSD. This article focuses not only on the differences between the functions of the FreeBSD and Linux kernels but also on the differences in their internal structures and interface specifications. It explains the challenges and techniques for implementing the same functions as those of the CRIU tool on FreeBSD.

1　はじめに

スマートフォンの普及や IoT の活用拡大にともなうインターネットユーザの増大に合わせて，Web サービス基盤に対するアクセス性能と可用性への要求が厳しくなっている．商用サービスでは，負荷に応じてサーバを追加投入するスケールアウトや，より高性能なハードウェアへ置き換えるスケールアップにより対応することが一般的となっている．一方，スケールアウトやスケールアップには，ハードウェアの購入や買い替え，さらに構築というコストがかかり，オンプレミスのサーバや小規模な Web サービス基盤では，これらの対応を行うことが難しい場合がある．

著者らは，ネットワーク通信など同様の機能が利用可能な複数の OS の間でも，その設計方針や実装の違いにより，性能特性が異なる可能性があることに着目

A Mechanism to Migrate Linux-compatible Processes on FreeBSD for Dynamic OS Switching

Katsuya Matsubara, Yuhei Takagawa, 公立はこだて未来大学, Future University Hakodate.

• 状況に応じた可用性と応答性能のバランス

• OSの脆弱性をつく攻撃の回避

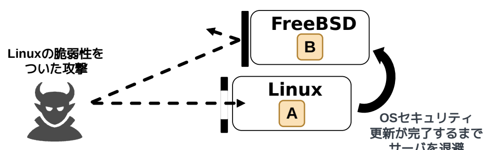

図 1　動的 OS 切り替えのユースケース

した．デスクトップ PC やスマートフォン，エンタープライズサーバなど，Windows や Linux などのいくつかの OS で寡占状態にあるプラットフォームも存在するが，特定の OS が独占しているプラットフォームは存在せず，現在でも，システムの目的や用途，プラットフォームの特性などに応じて複数の OS を使い分けている現状がある．そこで，本研究では，特性の異なる複数の OS を状況に合わせて動的に切り替えることで，OS 上で稼働するサービスの負荷耐性や性能の改善を試みる手法を提案している [6]．図 1 に示すユースケースのように，システム稼働中の動的な OS 切り替えが可能となることで，OS 上で実現されているサービスの可用性向上に効果的なソリューションが実現可能となる．前者のユースケースでは，OS のネットワーク性能特性の違いを活用することで，低負荷時の応答性能のメリットを享受しつつ，高負荷時には負荷耐性の高い OS へ動的に切り替えることで，処理性能と可用性のバランスをとることができる．後者のユースケースでは，OS 依存の脆弱性が発覚してシステムを更新する必要がある場合に，脆弱性のない別 OS に動作中のサービスを移行させることで可用性を維持できる．

本研究では，Linux と FreeBSD を対象として，サービス稼働中の状況に応じた複数 OS の動的な切り替えを実現するために，両 OS 間で相互運用できるプロセスマイグレーションを実現する．特に，Linux 上で動作中のプロセスを FreeBSD へ移送して実行継続するために，FreeBSD で対応する Linux 互換プロセスを対象としたマイグレーション機構を実装する．Linux のプロセスマイグレーションにおける標準ツールである CRIU を用いて作成したプロセスのチェックポイントを，FreeBSD 上の Linux 互換プロセスとして，その実行状態をレストアする．本論文では，主にプロセスマイグレーション機構の実装にフォーカスして，FreeBSD カーネルと Linux カーネルが提供する OS 機能の違いだけでなく，内部構造やインタフェース仕様の違いにも着目し，Linux CRIU の各機能と同等の機能を FreeBSD 上で実現する際の課題とそれに対処した実装の詳細を解説する．

以降，第 2 章では，実システムにおける動的 OS 切り替えの実現可能性を明らかにするために，稼働中の Web サーバを対象とした可用性向上の評価実験結果について示す．第 3 章では，プロセスマイグレーションに関する先行技術と関連研究について紹介する．第 4 章では，異種 OS 間のプロセスマイグレーションを実現する上での技術的課題をあげつつ，Linux-FreeBSD 間のプロセスマイグレーション実現へのアプローチ，および，FreeBSD 上での Linux 互換プロセスマイグレーション機構の実装詳細について述べる．最後に，第 5 章で，まとめと今後の課題について述べる．

2　動的 OS 切り替えのフィジビリティ評価

著者らの先行研究 [6] において，動的 OS 切り替えを用いた Web サービスの可用性向上に関する実現可能性について，実験を用いた評価を行った．本章では，動的 OS 切り替えの有用性を明らかにするために，本評価結果の要約を示す．

Linux と FreeBSD のそれぞれの OS 環境上で動作する Web サーバ Nginx に対して Apache Benchmark を用いてアクセス負荷をかけたときの性能について測定した．表 1 に示す仕様の PC 2 台のうち，1 台の PC で Apache Benchmark を実行し，ネットワークに接続されたもう 1 台の PC で FreeBSD または Linuxw 上で Nginx を稼働させた．Linux カーネルと FreeBSD カーネル，Nginx サーバは，それぞれ表 2，表 3，表 4 に示す設定を行った．また，Apache

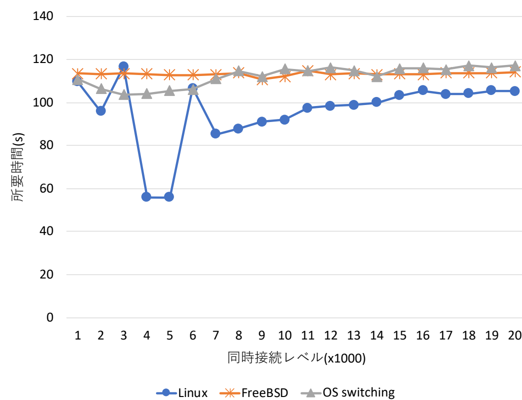

図 2　**Apache Benchmark における 100 万リクエストの処理に要した時間**

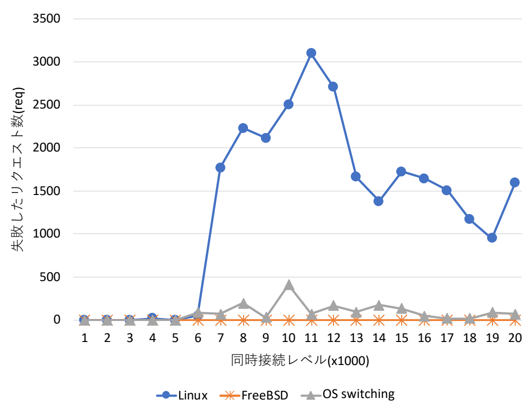

図 3　**Apache Benchmark 実行時における同時接続レベル別接続エラー発生数**

Benchmark は，Web サーバとの接続エラーが発生しても実行を中断しないように設定した．

図 2 は，Apache Benchmark から送信した全リクエストに対するレスポンスを受信するまでに要した所要時間を測定した．横軸は，Apache Benchmark で指定した同時接続数の目標値（以下，同時接続数レベル）を示し，縦軸は，全 20 万リクエストを送信して，それらに対するレスポンスをすべて受信するのに要した所要時間である．100 万リクエストの総処理時間について Linux 上と FreeBSD 上の結果を比較すると，Linux の一部の結果で応答時間に揺れが生じているが，すべての同時実行レベルにおいて Linux の方が FreeBSD よりも速いことが確認できる．

図 3 は，同時接続数レベルを変化させたときに，レスポンスを正しく受信できずに完了できなかったリクエストの数を示している．Linux の結果では，同時実行レベルが 6000 を超えると，一部のリクエストが送信に失敗していることがわかる．一方，FreeBSD では，同時実行レベルが変化しても，Apache Benchmark が送信したすべてのリクエストを処理できている．

前述の予備実験結果から，Linux と FreeBSD では，ネットワーク通信性能に関して異なる特性を持ってい

表 1　実験環境

CPU	Intel Core-i3 2.4GHz
RAM	4GB
Disk	SATA 5400RPM HDD
NIC	Intel 82579V Gigabit Ethernet
OS	CentOS 7.7-1908 (Linux kernel: 3.10)
	FreeBSD 13.0-CURRENT
Web Server	Nginx v1.16
Client App.	ApacheBench v2.3

表 2　**Linux カーネルのパラメータ設定値**

ulimit open files	113958
net.core.somaxconn	20000
net.ipv4.tcp_keepalive_time	7200
net.ipv4.tcp_tw_reuse	1
net.ipv4.ip_local_port_range	10000 65535
net.ipv4.tcp_max_syn_backlog	20000

表 3　**FreeBSD カーネルのパラメータ設定値**

ulimit open files	113958
kern.ipc.maxsockets	126627
kern.ipc.soacceptqueue	20000

表 4　**Nginx の設定**

worker_processes	1
worker_connections	20000
sendfile	off
keepalive_timeout	65
backlog	65535

ることが確認できた．FreeBSD のネットワーク機能は，同時接続数の増加などの高負荷状態に対して耐性がある．一方，Linux のネットワーク機能は高負荷耐性という点で FreeBSD に劣る部分があるが，処理性能の面では FreeBSD より優れていると判断できる．

前述までの Linux と FreeBSD のネットワーク通信特性を考慮して，Linux から FreeBSD への動的 OS 切り替えによる Web サーバの性能への影響を計測した．Linux 環境上で動作する Nginx プロセスに対して，Apache Benchmark からのリクエスト送信を開始してから 30 秒後にその実行中プロセスのチェックポイントを作成し，ネットワーク経由で転送した後，別の FreeBSD 環境上でその Nginx プロセスの実行状態を復元して Apache Benchmark からのリクエストを継続受信できるようにした．本測定では，プロセス移行時に転送される確立済 TCP コネクション数を最小限に抑えるために，Apache Benchmark における HTTP KeepAlive の使用を無効にし，さらに，プロセスのチェックポイント開始直前に到着する SYN パケットをドロップするパケットフィルタを有効にしている．図 2 と図 3 の系列”OS switching”は，Linux から FreeBSD へ OS を切り替えた場合の総所要時間と接続エラー発生数である．Linux と FreeBSD の結果の中間に分布しているこれらの計測結果から，OS 切り替え前後でそれぞれの稼働 OS に応じた性能特性を享受できており，OS 切り替えによってサーバの可用性や性能の特性を動的に変化させることは可能であることが確認できた．

3　先行技術

プロセスマイグレーションに関連する既存技術について説明する．

3.1　CRIU (Checkpoint/Restore in User-space)

Linux におけるプロセスマイグレーションの標準ツールとして知られている CRIU [1] [2] は，実行中のプロセスの状態を保存するチェックポイント機能と，保存したプロセスの実行状態を復元して実行を再開するレストア機能を実現している．CRIU の最大の特徴は，プロセスマイグレーションに必要な機能の多くをユーザ空間で実装することで，OS カーネルへの追加変更を最小限にしている点である．ネットワーク通信状況やファイルアクセス状態など OS カーネル内で管理されるプロセス関連情報の取得と復元に必要とされるプリミティブな機能のみを OS カーネルに追加実装している．

Linux では，ユーザ空間で動作するプロセスに対して，他のプロセスの内部状態を取得したり変更するような操作を制限している．CRIU では，パラサイトコードと呼ばれる，ptrace システムコールを用いて他プロセスから任意のプログラムをプロセスのメモリへ書き込んで実行させる手段を用いることで，ユーザ空間では外部からの操作が困難な，ファイルアクセス情報やネットワーク通信などの状態の取得と復元を実現している．

プロセスの実行状態の取得は，ptrace システムコールと procfs 仮想ファイルシステムを用いる．例えば，プロセス内のコードを実行している CPU レジスタの値は，ptrace システムコールの GETREGS を用いて取得できる．CPU 実行状態のレストアでは，割り込み処理が呼ばれた際に自動的にセットされる rt_sigreturn システムコールを利用して，取得したレジスタ値を CPU レジスタへ設定する．プロセスのメモリマップに関する情報は，procfs の maps ファイルから取得する．メモリの内容は，対象プロセス内で vmsplice システムコールと splice システムコールを用いて取得し，プロセス間通信機構であるパイプを用いてプロセス外へ出力する．メモリのレストアは，mmap システムコールを利用して，メモリ内容を保存したファイルをプロセスのメモリ空間にマップする．

CRIU では，メモリとソケットバッファ以外のプロセスの実行状態を，標準的なデータ形式のひとつである Google Protocol Buffer [4] 形式で保存する．メモリとソケットバッファの内容は，バイナリ RAW データ形式のまま保存する．

3.2　FreeBSD VPS (Virtual Private Sys-

表 5　**Linux と FreeBSD における open システムコールの差異**

	Linux	FreeBSD
システムコール番号	2	5
パラメータ値 O_CREAT の定義	0x200	0x0040

tem)

FreeBSD VPS[7] は，FreeBSD におけるプロセスマイグレーション機構のひとつである．FreeBSD Jail の拡張機能として実装されており，VPS インスタンスと呼ばれるプロセス隔離環境を単位として，FreeBSD 間でのマイグレーションを実現している．前述の CRIU とは異なり，プロセスマイグレーションに必要な機能のほとんどを OS カーネル内で実装している．

FreeBSD VPS におけるプロセス実行状態の取得処理では，ユーザ空間のメモリ内容だけでなく，OS カーネル空間のメモリも取得している．カーネル空間には，プロセスごとのカーネルスタックが存在しており，このカーネルスタックを取得することができる．CPU の情報は，カーネルスタックの先頭部分に存在する PCB（Process Control Block，プロセス制御ブロック）を利用する．PCB は，プロセスが待ち状態になった場合に，処理が中断されても，続きから再開できるように，CPU レジスタの情報がカーネルによって格納される．

プロセスのレストア時は，メモリ内容を復元する際に合わせて復元されるカーネル空間内の PCB を利用し，実行可能プロセスとして OS スケジューラに登録することで実行を再開する．

3.3　FreeBSD の Linux バイナリ互換機能

Linux バイナリ互換機能[3][5] は，Linux ELF 形式のプログラムファイルを FreeBSD 上で実行できるようにする機能である．Linux と FreeBSD の両 OS において，同等な機能を持つシステムコールが数多くで提供されておるが，同じ機能のシステムコールにおいても，システムコール番号や引数パラメータが異なる場合がある（表 5 参照）．そのため，本研究が目的とする異なる OS 間でプロセスに含まれるプログラムコードとその実行状態をマイグレーションして実行を再開しても，移行先のカーネルの機能を正しく呼び出すことができない場合がある．

FreeBSD の Linux バイナリ互換機能は, Linux ELF 形式のプログラムコードに含まれるシステムコールに対して，その実行時にシステムコールテーブルの順番変更や引数パラメータの値の変換を行い，対応する FreeBSD のシステムコールを利用できるようにしている．Linux バイナリ互換機能は，プロセスのメモリへ実行可能ファイルの内容がロードされる際に，Linux ELF 形式であるかどうかを判別する．Linux ELF の場合には，対応するプロセス向けの OS カーネル内のシステムコールテーブルを変更し，Linux 互換用システムコールテーブルに差し替える．プロセスから Linux のシステムコールが実行された際, FreeBSD カーネル内部に実装された Linux システムコール関数が呼び出され，その関数内で対応する FreeBSD カーネル機能を呼び出す．また，Linux のシステムコール引数として渡されるパラメータ値は，OS カーネル内で FreeBSD で処理できるパラメータに変換する．

前述した全ての処理が OS カーネル内で行われるため，ユーザ空間のプロセスのメモリ内容は，スタックを含めて Linux で実行した場合と同じ内容となる．

4　Linux-FreeBSD 間プロセスマイグレーションの実装

前章で述べたとおり，Linux と FreeBSD のそれぞれには，同一 OS 内のプロセスマイグレーションを実現する機能が存在する．また，FreeBSD には，Linux ELF 形式のプログラムを実行するためのバイナリ互換機能が備わっている．しかし，Linux と FreeBSD の既存プロセスマイグレーション機能はその設計や実装が大きく異なり，それぞれで作成したプロセスチェックポイントを相互運用することはできない．本研究では，Linux-FreeBSD 間でのプロセスマイグレーションを実現するために，相互運用可能なプロセスのチェックポイント／レストア手法を確立する．

Linux-FreeBSD 間で相互運用可能なプロセスマイグレーションを実現するアプローチとして，FreeBSD VPS のチェックポイント／レストア機能や入出力形式に対応するように Linux CRIU を改造する手法と，

Linux CRIU のチェックポイント／レストア機能や入出力形式に対応するように FreeBSD のプロセスマイグレーション機構を実現する手法が考えられる．前者の場合，FreeBSD VPS が OS カーネル内で実装されているため，カーネルの内部データ形式に合わせて Linux カーネルや CRIU への大幅な変更を必要とする可能性がある．後者の場合，CRIU がもつ機能の多くがユーザ空間で実装されているため，FreeBSD 上での実現が比較的容易であると考える．これらの考察から，FreeBSD 上の Linux 互換プロセスを対象として，Linux CRIU と相互運用可能なプロセスマイグレーション機構を実装することとした．

以降，FreeBSD 上における Linux 互換プロセスのチェックポイント／レストア実現に必要な機能について，それぞれの技術的課題と実装における解決手法の詳細を述べる．

4.1 システムコール・インジェクション

外部からプロセスに関連する内部情報を取得したり設定・制御できる procfs をもつ Linux と比較して，FreeBSD では外部からプロセスを制御できる範囲により制約されている．本実装では，CRIU におけるパラサイトコードと同様の技術を活用して，対象プロセス自身にプロセスに関する制御を行わせる手段として，システムコール・インジェクション機能を実現した．ptrace システムコールを用いて，システムコールを呼び出すための命令である syscall 命令 (0x050f)，および，実行後にトレースを再開するためのブレイクポイントである int3 命令 (0xcccc) を対象プロセスの空きメモリへ書き込む．さらに，表 6 で示す各レジスタに対して，システムコール番号と引数を設定する．

4.2 プロセス実行状態の移送

プロセス実行状態とは，CPU レジスタとメモリの状態である．復元時には，対象プロセスを作成するために，fork システムコールと execve システムコールを用いる．図 4 は，CPU レジスタとメモリの状態を復元する概要図である．

表 6　システムコール呼び出し時のレジスタ

レジスタ	システムコール
RAX	番号
RDI	第 1 引数
RSI	第 2 引数
RDX	第 3 引数
R10	第 4 引数
R8	第 5 引数
R9	第 6 引数

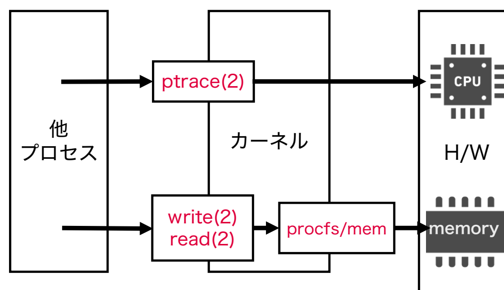

図 4　プロセス実行状態復元の概要図

4.2.1 プロセスの初期化

Linux バイナリ互換機能を利用するプロセスを復元する場合，復元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，ptrace システムコールの TRACEME を有効にすることで，対象プロセスのエントリポイントをフックし復元処理を行う．しかし，Linux バイナリ互換機能を利用するプロセスでは，Linux 互換機能を有効にする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，プロセスの状態を復元する．なお，本研究では，main 関数の開始命令のアドレスをバイナリ解析を行うこ

	0	4	8	12–15 byte
Linux	r15	r14	r13	r12
	rbp	rbx	r11	r10
	r9	r8	rax	rcx
	rdx	rsi	rdi	orig_rax
	rip	cs	eflags	rsp
	ss	fs_base	gs_base	ds
	es	fs	gs	
FreeBSD	r15		r14	
	r13		r12	
	r11		r10	
	r9		r8	
	rdi		rsi	
	rbp		rbx	
	rdx		rcx	
	rax		r_trapno / r_fs / r_gs	
	r_err / r_es / r_ds		r_rip	
	r_cs		r_rflags	
	r_rsp		r_ss	

図 5　Linux と FreeBSD のレジスタ構造体

とで取得している．

4.2.2　CPU レジスタ値の移送

取得時には，ptrace システムコールの GETREGS を用いてレジスタの情報を格納する構造体 (以下，レジスタ構造体) を利用する．レジスタ構造体には，汎用レジスタ，インデックスレジスタ，セグメントレジスタ，フラグレジスタの各値が格納されている．

復元時には，レジスタ構造体に取得したレジスタの情報を格納し，ptrace システムコールの SETREGS を利用する．

ABI が異なる Linux と FreeBSD では，ptrace システムコールで扱うレジスタ構造体が異なる（図 5 参照）．メンバ変数の順番の差異，サイズ，汎用レジスタ AX を表すメンバ変数の差異があるため，異なる OS で取得した構造体データを変換する必要がある．一般的に，汎用レジスタ AX には呼び出すシステムコール番号とシステムコール返り値が格納される．Linux では，レジスタ構造体に汎用レジスタ AX を表すメンバ変数として，orig_rax と rax が存在する．orig_rax にはシステムコール番号が格納され，rax にはシステムコール返り値が格納される．一方，FreeBSD では，レジスタ構造体に汎用レジスタ AX を表すメンバ変数は rax しか存在せず，システムコール番号とシステムコール返り値のどちらかが格納される．

異種 OS 間プロセスマイグレーションを実現するためには，FreeBSD のメンバ変数 rax に格納する値を Linux のメンバ変数 orig_rax と rax から選択する必要がある．次に実行すべき命令のアドレスを指すプログラムカウンタが示している命令が，syscall 命令[†1]であり，汎用レジスタ AX がシステムコール番号ではなく，システムコール返り値を指している場合，システムコール番号が異なり，正しく実行されず，プロセスが停止してしまうためである．判断はプログラムカウンタが指す命令からシステムコール実行の前後を判断して，システムコール実行前であれば，Linux のメンバ変数 orig_rax を格納し，システムコール実行後であれば，Linux のメンバ変数 rax を格納する．

また，OS が異なるでセグメントレジスタをそのまま復元する場合，セグメントのサイズや位置などが変わってしまい，復元後正しく動作しないため，Intel x86 アーキテクチャでは，レジスタ構造体の情報を復元する際に，セグメントレジスタ (cs,ss,es,fs,gs) は，OS がプロセス生成時に割り当てた値をする必要がある．本実装では，取得したセグメントレジスタの値を使用せず，レストア先の OS が割り当てた値をそのまま利用する．

4.2.3　メモリ内容の移送

CRIU では，vmsplice システムコールや splice システムコールを利用することで，パイプ経由でメモリ状態をファイルに保存していたが，FreeBSD には vmsplice システムコールや splice システムコールが存在しないため，CRIU と同様の方式でメモリの取得ができない．そのため，メモリ状態は，ptrace システムコールのアタッチ時に，procfs の mem ファイルを読み込むことで取得する．取得するメモリ内容は，プロセス実行中に値が変更されていく仮想メモリ空間におけるデータ領域やヒープ領域，スタック領域，mmap システムコールで新しく確保した領域を対象とする．プロセスマイグレーションは同じプログラムを実行することが前提であるため，テキスト領域や共有ライブラリは取得する必要がない．ただし，テキスト領域内部に含まれるライブラリ関数を参照するためのアドレスを保持しているテーブルは，ライブラリのロード時に変更されるため，取得しなければなら

†1　システムコール実行の割り込みを発生させるアセンブラ命令

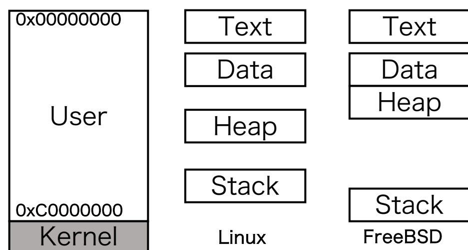

図 6　Linux と FreeBSD のメモリ領域の差異

ない．

これらの領域は，仮想メモリ上に連続しないように配置されるため，各領域のメモリ内容を取得するためには，仮想メモリの領域 (以下，メモリマップ) 情報を取得する必要がある．メモリマップ情報は，procfs の map ファイル，あるいは，libprocstat ライブラリを利用することで取得できる．本実装では，既存の構造体として情報を扱える libprocstat ライブラリを利用する．libprocstat ライブラリによって取得できる構造体 kinfo_vmentry には，その領域の開始アドレス，終端アドレス，アクセス権限などが格納されている．

復元の際には，メモリマップ情報をもとに，procfs の mem ファイルを介して，取得したメモリのデータ，ヒープ，スタックの各領域のメモリ状態を書き込む．

図 6 で示すように，OS が異なるヒープ領域やスタック領域，ライブラリの配置アドレスが異なる．この場合，メモリを復元する際に書き込みを正しく行えず，適切なメモリとして復元することができない．そこで，本実装では，図 7 で示すように，すでに配置されているメモリ領域を munmap システムコールを用いて解放し，mmap システムコールを用いて再度確保することで，正しいアドレスにメモリ領域が存在するようにする．なお，munmap システムコールと mmap システムコールに関しても，システムコールインジェクションを行う必要がある．

4.2.4　vDSO への対処

vDSO(virual dynamic shared object) は，gettimeofday システムコールや clock_gettime システムコールなどの頻繁に呼び出されるような特定のシステムコールを高速に実行するためのライブラリである．

図 7　メモリレイアウト変更の手順

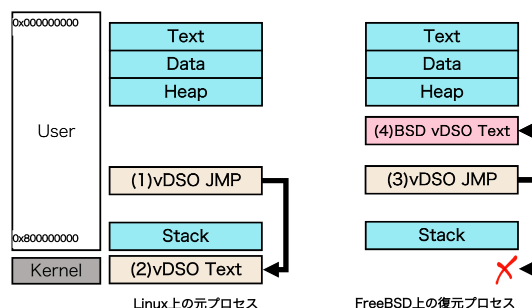

図 8　vDSO の差異

図 8 には，Linux と FreeBSD における vDSO の配置に関して示している．vDSO のテキスト領域は，Linux ではカーネル空間内 (2) に配置され，FreeBSD ではユーザ空間内 (4) に配置される．(1) の領域は，vDSO のテキスト領域へアクセスするための領域であり，カーネル内部に配置されている (2) にアクセスする．そのため，FreeBSD 上で (3) の領域を復元したとしても，カーネル空間を参照しているアドレスにはアクセスできないため，実行することはできない．そこで，本研究では復元した (3) の領域が (4) の vDSO のテキスト領域を指すように変更する．

4.3　ファイルアクセス状態の移送

プロセスがファイルにアクセスする手段には，open システムコールを用いてファイルディスクリプタを

介する場合と，mmap システムコールを用いて仮想メモリ内にマッピングする場合が存在する．

4.3.1 mmap システムコールによるファイルアクセス

mmap システムコールの場合には，ファイルがメモリにマッピングされ，他のメモリ領域と同様にアドレス参照による入出力が可能となる．取得時には，4.2.3 項で述べた方法と同様に，ファイルの内容をメモリの状態として取得するが，どのファイルディスクリプタと関連づけられていたかを取得する必要がある．

復元する際には，以下のようにメモリ上へのファイルマッピングと，4.2.3 項で述べたメモリの復元を行う．

1. 復元すべきフィイルを 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アクセスモードやフラグを設定
3. mmap システムコールを用いて，ファイルディスクリプタ番号を指定してメモリ上に配置する
4. close システムコールを用いて，open システムコールで開いたファイルを閉じる
5. procfs の mem ファイルに対して write システムコールでデータを書き込む

4.3.2 open システムコールによるファイルアクセス

open システムコールの場合には，ファイルに対して，ファイルディスクリプタと呼ばれる番号が振られる．read システムコールや write システムコールなどのファイルを扱うシステムコールは全てファイルディスクリプタを利用して読み込みや書き込みなどが行われる．プロセスマイグレーションに必要な情報は，オープンしているファイルのパス名，ファイルディスクリプタ，ファイルのシーク位置 (ファイルオフセット)，オープンモードやフラグである．ファイルに関する情報は，devfs の fd ディレクトリから取得することができる．しかし，devfs の fd ディレクトリはアクセスしたプロセスの情報しか表示しないため，他のプロセスからファイルアクセス情報を取得することはできない．そこで本実装では，他のプロセスの情報を取得できる libprocstat ライブラリを利用することでファイルに関する情報を取得する．libprocstat ライブラリはプロセスが持つ情報をまとめた構造体のリストを制御するライブラリで，特定のプロセスを PID によって指定することができる．libprocstat ライブラリによって取得できる filestat 構造体は，ファイルディスクリプタ，ファイルパス，ファイルオフセットなどを含む．

また，性能向上のためにファイルの書き込みは，遅延書き込み (または，writeback) で行われる．遅延書き込みでは，write システムコールを用いたとしても，一度 OS のバッファ (ページキャッシュ) に格納された後，同時に書込みが行われる．そのため，プロセスから見た書込みと実際のファイルへの書き込みは異なる場合がある．sync システムコールを用いることで，バッファ上のデータをファイルに書き込む．

復元時には，システムコールインジェクションを行い，以下の手順で復元する．

1. 復元すべきファイルを 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アクセスモードやフラグを設定
3. dup2/close システムコールを用いて，ファイルディスクリプタ番号を変更
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

4.4 ネットワーク通信状態の移送

ネットワーク通信状態である TCP コネクションの状態を復元することを 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．本研究では FreeBSD カーネルに TCP repair を移植するために，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 を行うためには，以下の要件を満たす必要がある．

TCP repair オプション追加

ネットワークに関する情報を扱う getsockopt シ

ステムコール/setsockopt システムコールを介して TCP コネクションに関するパラメータの取得と復元を行えるようにするため

送受信キューのデータの取得と反映

マイグレーション後に，送受信中のデータが損失し，正常なデータを送受信できなくなるのを防ぐため

シーケンス番号の取得と反映

マイグレーション後に，送受信パケットの位置がリセットされ，正常に送受信できなくなるのを防ぐため

接続の維持

マイグレーション時に，コネクションが切断され，再接続のために，通信する両者が接続処理を再度行わなければならないのを防ぐため

4.4.1 送受信キュー内のデータの移送

Linux では, getsockopt/setsockopt システムコールのオプションに, SO_REPAIR, SO_REPAIR_QUEUE, SO_QUEUE_SEQ を追加している．

SO_REPAIR オプションを無効にした場合，従来までの送受信処理と同様に．SO_REPAIR オプションを有効にした場合，送受信キューのデータの取得と反映，シーケンス番号の取得と反映，接続の維持が可能となる．SO_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．SO_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．

図 9 に示す番号はプロセスから送信された通信データの各状態を示しており，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．(2) と (4) のデータは通常であれば，ユーザ空間からは復元することのできないデータである．

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 システムコールに SO_REPAIR_QUEUE オプションを使うことで，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．

4.4.2 シーケンス番号の移送

TCP 通信において，シーケンス番号は送受信されたパケットの位置を示すため，シーケンス番号を正しく復元できない場合，TCP 通信が正しく行えなくなる．そのため，シーケンス番号を復元する必要がある．

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．シーケンス番号の取得と復元を行う setsockopt システムコールに SO_REPAIR_SEQ オプションを追加し，SO_REPAIR_QUEUE オプションを用いて指定したキューに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．

4.4.3 接続状態の維持・復元

TCP 通信において，ソケットを閉鎖させる場合，3 ウェイハンドシェイクにより，FIN パケットを送信し切断処理を行い，コネクションが切断されるため，再度通信をするためには接続要求である SYN パケットの送信が必要となる．この場合，クライアントは再接続を見越した実装が予めされていなければならない．そのため，ソケット閉鎖時に FIN パケットを送信せず，接続要求時に SYN パケットも送信しないよ

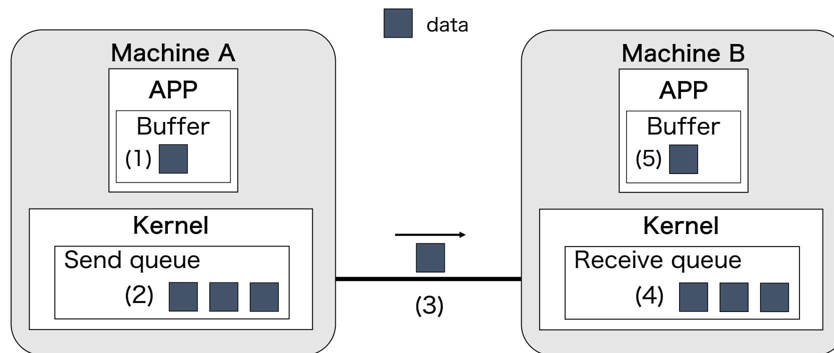

図 9　**TCP** 通信におけるデータの状態

うにすることで，実際には接続が切断されている状態でも，クライアント側は接続が切断されていない状態としてレスポンスを待つ状態を保つことができる．

TCP repair が有効になっている場合に，connect システムコールを SYN パケットを送信しないように変更し，close システムコールを FIN パケットを送信しないように変更する．

4.4.4　フロー制御関連パラメータの移送

OS が異なるとネットワーク・プロトコルスタックの実装が異なる．この場合，カーネル内部で利用しているネットワーク通信のための変数や条件式が異なるため，変数を正しく復元できず，条件式を満たすことができないことがある．条件式を満たすことができなければ，データの送受信に影響がある．特に本論文の対象としている Linux から FreeBSD にプロセスマイグレーションをする際には，フロー制御が行われず，ウィンドウサイズが減少し続け，通信できなくなる可能性がある．FreeBSD 上でウィンドウサイズが更新されなくなるのは，送信キューのウィンドウの開始位置である SND.WL2 を Linux では利用しておらず，取得されないため，適した値を SND.WL2 に代入できないからである (図 10 参照)．

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表 11 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) が真になる場合は全て更新されない．

本研究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) を満たすようにする．つまり，復元時には 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，変更しても問題がない．

4.5　イベントハンドラ登録の移送

イベントハンドラとは，シグナルや I/O などのイベントがあった際に，カーネルがイベントに応じた処理をプロセスに実行させるものである．例えば，Nginx では I/O イベントを用いて，HTTP 接続を要求されるまでプロセスを待機状態にすることで，プロセスが使うリソースを削減する．また，シグナルイベントを用いて，設定ファイルの更新やログファイルの開き直しなどを行う．シグナルイベントは，sigaction システムコールを実行することでカーネルに登録され，I/O イベントは，epoll システムコール群を用いることで OS カーネルにイベントを登録する．さらに，epoll_create システムコールで専用のインスタンスを作成し，epoll_ctl システムコールでイベントを OS カーネルに登録できる．

本実装では，システムコールインジェクションを用いて sigaction システムコールや epoll システムコールをレストア対象のプロセス内で実行させることで，マイグレーション先の OS カーネルにイベントハンドラを再登録する．

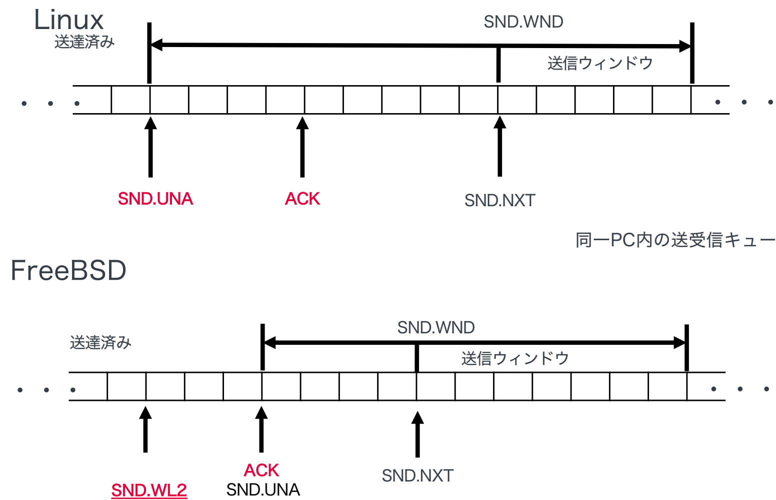

図 10　Linux と FreeBSD のフロー制御

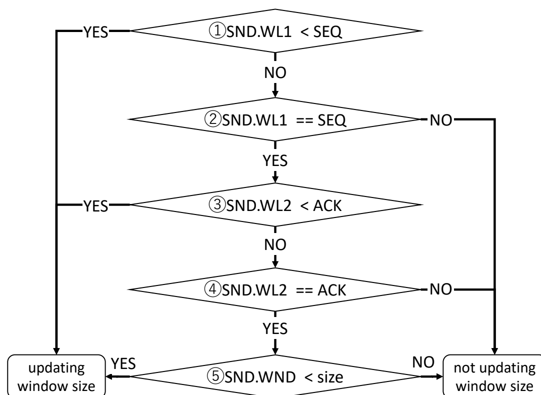

図 11　ウィンドウサイズ更新のフローチャート

epoll システムコール群は，Linux にしか存在しないシステムコールであるが，FreeBSD の Linux バイナリ互換機能によって同等のシステムコール機能が実現されている t ため，FreeBSD における Linux プロセスのレストアに際して特段の対処は必要ない．

5　おわりに

本論文では，Linux と FreeBSD の間での動的な OS 切り替えを目的として，FreeBSD 上での Linux 互換プロセスのチェックポイント機能，および，Linux 上で実行中のプロセスに対して CRIU を使って作成されたチェックポイントを FreeBSD 上でレストアする機能の実現に際して，技術的課題と対処手法，FreeBSD 上での実装詳細をとりまとめた．Linux と FreeBSD 間のプロセスマイグレーションを実現するために，Linux のプロセス形式を共通形式とし，FreeBSD では Linux バイナリ互換機能により Linux プロセスの実行を可能にする．マイグレーション技術についても Linux の CRIU を標準技術とし，FreeBSD 上で CRIU ライクなプロセスマイグレーション機構を実現した．ネットワーク通信状態のマイグレーションを可能にするために，TCP repair 機能を FreeBSD に実装した．Linux カーネルで実装されている FreeBSD の Linux バイナリ互換機能が対応できない Linux と FreeBSD の差異については，メモリレイアウトや vDSO，ネットワーク・プロトコルスタックのそれぞれに対して OS 毎の差異に対処する方法を検討して実装した．

今後の課題として，OS 切り替え契機の判断をより適応的に行う仕組みの考案がある．第 2 章で示した評価実験では，固定されたタイミングでプロセスマ

イグレーションを行っていた．状況の変化に対して，マイグレーションオーバヘッドも考慮しつつ OS 切り替えをすべきかどうかを柔軟に判断する手法を確立する．また，プロセスマイグレーション時のオーバヘッドとなる，チェックポイントしたプロセス情報の転送最適化を行いたい．本実装では，チェックポイント処理によりプロセス状態をファイルに一旦保存した後，ファイルシステム間でネットワークコピーした後にレストア処理を開始していたが，プレコピーやポストコピー技術を導入することで，プロセスマイグレーション時のダウンタイムの削減を試みる．さらに，ソフトウェア流通の単位として近年急速に普及しているコンテナを単位とした Linux-FreeBSD 間マイグレーションの実現がある．コンテナを実現する計算資源の隔離や制限機能について Linux と FreeBSD のそれぞれの対応機能を精査した結果 [8] にもとづいて，Linux-FreeBSD 間のコンテナマイグレーション機構を実現したい．

参考文献

[1] A. Mirkin, A. Kuznetsov and K. Kolyshkin: Containers checkpointing and live migration, *Proceedings of the 2008 Ottawa Linux Symposium*, Vol. 2, 2008, pp. 85–90.

[2] CRIU: Checkpoint/Restart in Userspace.

[3] Divacky Roman: Linux® emulation in FreeBSD.

[4] Google Inc.: Protocol Buffers.

[5] Jim Mock: Chapter 10. Linux Binary Compatibility, `https://www.freebsd.org/doc/handbook/linuxemu.html`.

[6] Matsubara, K. and Takagawa, Y.: Adaptive OS Switching for Improving Availability During Web Traffic Surges: A Feasibility Study, *Proceedings of The 8th IEEE Internat. Workshop on Architecture, Design, Deployment & Management of Networks & Applications (ADMNET 2020)*, 2020, pp. 1176–1182.

[7] Ohrhallinger, K. P.: Virtual Private System for FreeBSD, *EuroBSDCon 2010*, 2010.

[8] Takagawa, Y. and Matsubara, K.: Yet Another Container Migration on FreeBSD, *AsiaBSDCon 2019 Proceedings*, 2019, pp. 97–102.