Vol.2014-OS-130 No.23
2014/7/29

多重OS構成によるカーネルのライブアップデート手法の提案

石川 幸希[1,a)]　安井 裕亮[1]　齋藤 彰一[1]　瀧本 栄二[2]　毛利 公一[2]　松尾 啓志[1]

概要：計算機やOSの高機能化に伴い増加する脆弱性に対応するため, 定期的なアップデートによりOSを最新の状態を常に保つ必要がある．OSアップデートは通常, 最新版OSの適用, リブートを経て完了するが, リブートを伴うOSアップデートはWebサーバ機能を担うような計算機の可用性を著しく低下させるといった問題があり, アップデートに係るダウンタイムの最小化が求められている．そこで本稿では, リブートを必要としないOSのアップデート手法を提案する．OSをアップデートする上で問題となるOSバージョン間の内部情報の差異を吸収することで, バージョンに依存しない内部情報のマイグレーションを実現し, OSのアップデートを可能とする．本稿ではOSフェイルオーバー機構Orthros上で起動させた異バージョンOS間でファイルキャッシュをマイグレーションし, アップデートする手法について述べる．

キーワード：OS，ライブアップデート，ダウンタイム短縮，ファイルキャッシュマイグレーション

A Proposal for Live Update of Linux Kernel on Multiple OSes Execution Platform

Koki Ishikawa[1,a)]　Yusuke Yasui[1]　Shoichi Saito[1]　Eji Takimoto[2]　Koichi Mori[2]
Hiroshi Matsuo[1]

1. はじめに

ソフトウェアに潜む脆弱性は，個人情報の流出やウィルス感染，外部からの不正アクセスにつながるため，定期的なアップデートによりソフトウェアを最新の状態に保つことが重要である．ハードウェアとプロセスを含めた計算機全体の動作を管理するOSカーネルも例外ではなく，複雑ゆえに脆弱性が多数報告され，比較的短い間隔でそれらの脆弱性を排除した新たなバージョンのOSカーネルがリリースされている [1].

通常，ソフトウェアのアップデートでは，新バージョンのインストールと計算機の再起動という2つの処理が必要である．ブラウザやメールクライアントといったユーザレベルのソフトウェアの場合には当該プロセスのみの再起動で十分となる場合が多いが，OSカーネルの場合は計算機起動時にメモリにロードされたコードが処理されることから，計算機の再起動が必要となる．このため，新しいバージョンのOSカーネルのインストールと計算機の再起動により，OSカーネルの停止から動作再開までに場合によっては数分を要し，その間は計算機によるアプリケーションの処理は停止する．

Webサーバやデータベースサーバのような対外サービスを運用する計算機では，ダウンタイム増加による可用性の低下が問題となるため，再起動を伴うアップデートの実施は必要最低限とする必要がある．また，再起動によりOSカーネル停止直前のアプリケーションの実行状態が失われるため，起動直後のサービスはゼロ状態から処理が始まり，サービスの継続運用に問題が生じる．このようなサービス品質の面からも，計算機の再起動の実施は望ましくない．

ダウンタイムを短縮する方法として，デバイスの初期化

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University
[a)] orthros@mail.ssn.nitech.ac.jp

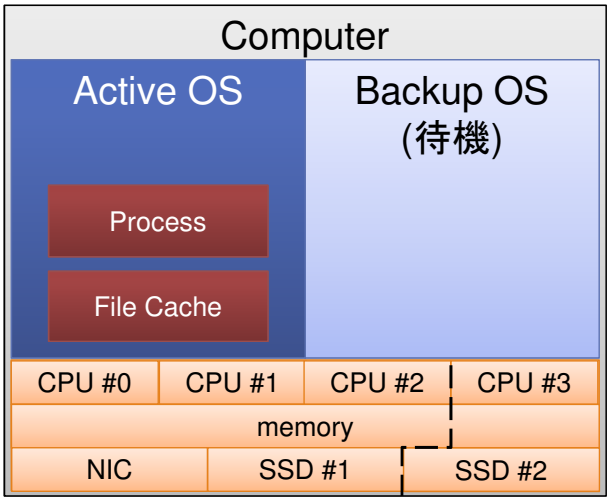


図 1　Orthros の構成図

を省いたウォームブートを実現する Kexec[2] がある．しかし，Kexec も数秒から数十秒を復帰に要する上，実行状態の損失は避けられず，抜本的な問題の解決には至らない．

そこで，本稿では計算機の再起動を伴わない，OS カーネルのライブアップデート手法について述べる．ライブアップデートにおいて問題となる OS のバージョンに依存するデータ構造の差異を吸収し，多重 OS 機構を用いて，ファイルキャッシュやプロセスをアップデート前の OS カーネルから最新版の OS カーネルへマイグレーションすることで再起動を必要としない OS のアップデートを実現する．これにより，ダウンタイムの最小化だけでなく実行状態の損失の防止が可能となり，可用性の低下とサービス品質の低下の双方の抑制を実現する．

まず，2 章で提案手法の基盤となる多重 OS 機構 Orthros について述べる．次に，3 章では提案手法の概要に触れ，4 章で提案の設計と実装について述べる．5 章で評価を行い，6 章で関連手法について述べる．最後に，7 章でまとめを述べる．

2. Orthros

本稿は，多重 OS 機構を用いたカーネルアップデートについて述べる．本章では，提案手法の基盤となる多重 OS 機構の Orthros[3] の概要と Orthros に実装されている機構について述べる．

2.1 概要

Orthros は単一の計算機のハードウェアリソースをソフトウェアレベルで論理的に分割する Softwear Logical Partitioning(Software LPAR)[4] を採用し，OS を多重起動することでフェイルオーバーを実現する耐障害性向上を目的としたシステムである．**図 1** に Orthros の構成の概要を示す．ユーザとシステムサービスの主な仕事を処理する OS を ActiveOS とし，ActiveOS に障害が発生した時のために待機している予備の OS を BackupOS とする．BackupOS は通常時は何もせず，最低限のハードウェア構成で動作する．Orthros では以下のような工程でフェイルオーバーが行われる．

(1) あらかじめ BackupOS を起動する．
(2) ActiveOS に障害が発生する．
(3) 障害を検知した BackupOS が，ActiveOS によって使用されていたデバイス（SSD，NIC）をマイグレーションする．
(4) ファイルキャッシュをマイグレーションする．
(5) プロセスをマイグレーションする．
(6) NIC に IP を割り当てる．

(1) では ActiveOS 上で BackupOS をメモリへロードし，単一コア上で起動する．BackupOS の起動処理は ActiveOS の起動スクリプト中で実施される．(3) から (5) の処理は，BackupOS 上で動作するフェイルオーバー用のプロセスによって行われ，ユーザに対する操作は求められない．Orthros では上記工程のようなフェイルオーバーを実現するにあたり，デバイスマイグレーション機構，ファイルキャッシュマイグレーション機構，プロセスマイグレーション機構，死活監視機構が実装されている．特殊なハードウェアを必要としないため，導入コストが抑えつつ，耐障害性の向上を実現している．

2.2 デバイスマイグレーション機構

ActiveOS と BackupOS は予め占有使用するハードウェアをそれぞれ指定しており，それぞれの OS が起動するときには使用対象のハードウェアのみを初期化し使用する．使用しないデバイスは後から初期化処理ができるよう，初期化パラメータを保持しておく．ActiveOS に障害が発生した時，BackupOS のフェイルオーバー用プロセスが，未初期化のデバイス，すなわち ActiveOS が使用していたデバイスの初期化をする．初期化によって BackupOS で ActiveOS が使用していたデバイスの認識を行うことで，デバイスマイグレーションは実現されている．デバイスマイグレーションでは，ActiveOS の環境を BackupOS 上でなるべく再現するため，ActiveOS が使用していた NIC とディスクがマイグレーションされる．

2.3 ファイルキャッシュマイグレーション機構

ファイルキャッシュは，ディスクから読み出したデータを主記憶上でページ単位で管理される．キャッシュされたページに変更があると，同期もしくは非同期にディスクに書き込まれ，データの一貫性が保たれる．同期による反映では，書き込み要求があると即座にディスクに書き込まれるため，ファイルの一貫性が欠如しないというメリットがあるが，入出力処理の回数が増えるためアクセス速度が遅くなる．一方，非同期による反映は，変更をまとめてディ

スクへ書き込むため，同期による反映と比べて入出力回数の削減が見込まれ効率的である．しかし，この場合，ディスクへの書き込み前に OS カーネルが停止すると，メモリ上のキャッシュが失われ，データの一貫性が失われる可能性がある．これらのファイルキャッシュを保護する機構が，ファイルキャッシュマイグレーション機構である．

ファイルはユーザデータだけではなくメタデータも含めて inode 構造体で管理され，一つのファイルにつき一つの inode 構造体が存在する．ファイルキャッシュはファイルシステムごとに存在する super_block 構造体を起点とした木構造で管理されており，super_block 構造体からディスクに書き込まれていない dirty なページを持つファイルの inode 構造体のリストを辿ることができる．

Orthros では，BackupOS で動作するフェイルオーバー用プロセスが，ActiveOS のメモリ領域に残留するファイルキャッシュを検索，dirty なページを BackupOS 上のメモリへコピーし，ActiveOS 上で使用されていた時と同じパスでマウントされたディスクの inode 構造体に関連付けることで保護する．

2.4 プロセスマイグレーション機構

ActiveOS で事前に指定された pid-namespace に含まれるプロセス群を対象とし，BackupOS が ActiveOS のメモリイメージからプロセス管理構造体を読み出した後に BackupOS 内で再構成する．fork システムコールが基盤となっており，新たにプロセスを生成後に，ActiveOS 上の停止したプロセスの実行状態をコピーすることで，マイグレーションを実現している．プロセスがオープンしていたファイルの内容は，ファイルキャッシュマイグレーションにより保護されているため，プロセスが持つファイルディスクリプタの状態を引き継げばファイルの状態も保持される．

2.5 死活監視機構

BackupOS は Inter-Processor Interrupt（IPI）による ActiveOS の死活監視を行う．ActiveOS が正常に動作していることを伝える生存通知と，ActiveOS に障害が起きたことを通知する異常通知の 2 種類が使われる．定期的に BackupOS に対して生存通知として IPI を送信し，生存を通知する．BackupOS は一定時間この生存通知を受信しない場合に ActiveOS が異常停止したと認識する．また，ActiveOS が自身の障害を検知して panic() 関数が呼ばれたときは異常を通知する IPI が送信され，BackupOS は ActiveOS に障害が発生したことを検知する．

3. 提案手法

本稿では，Software LPAR による多重 OS 機構を用いた再起動を伴わない OS カーネルのライブアップデート手法を提案する．本手法では，新しいバージョンの OS を別途起動した上で，プロセスの実行状態を古い OS から新しい OS に移行する．これによって、プロセスが動作する OS のバージョンの更新と，プロセスの実行の継続を実現する．

なお，本稿では，アップデートを限りなく繰り返すことのできるシステムの開発を目標としているため，初めに起動した OS を 1st OS，2 番目に起動した OS を 2nd OS と呼び，N 番目に起動した OS を Nth OS と呼ぶこととする．以降ではカーネルバージョンの違いによるアップデートに焦点を当てるために，1st OS から 2nd OS へのアップデートを対象として述べる．

3.1 必要要件

多重 OS 機構を用いたライブアップデートを実現するための要件を以下に定める．

- ユーザによる 1st OS の任意時点での停止
- 2nd OS による 1st OS の停止検知
- OS 間でのデバイスマイグレーション
- OS 間でのファイルキャッシュマイグレーション
- OS 間でのプロセスマイグレーション

ユーザによる 1st OS の任意時点での停止は，ユーザビリティを下げないための要件である．アプリケーションを停止させるようなアップデートにむけた環境整備をユーザに強いるのは，ユーザビリティを下げる大きな原因である．そのため，ユーザは停止する時点や環境を意識すること無く，任意の時点でアップデート処理をできるようにする．

2nd OS による 1st OS の停止検知は，1st OS から 2nd OS へスムーズに処理を移すためには必要である．これを実現しない場合，手動で 2nd OS に対し 1st OS の停止を知らせ，アップデート処理を呼び出す必要がある．これは，停止とアップデート開始までにタイムラグが生まれ，ダウンタイムの増加を招く原因となる．

OS 間でのデバイスマイグレーションは，アップデート前とアップデート後で動作環境を大きく変えないために必要である．

OS 間でのファイルキャッシュマイグレーションは，1st OS の任意時点での停止を要件とした場合に必要となる．なぜなら，ディスクに書き込まれていないファイルキャッシュがメモリに存在している時点で 1st OS を停止した場合，そのファイル情報は失われ，アップデート後に失ったファイル情報を参照できないためである．

OS 間でのプロセスマイグレーションは，アップデートによるサービス品質の低下を抑制するための要件である．プロセスの再起動は，サービスの一時的な停止を伴い，停止時点で処理途中であったサービスの異常停止を招く恐れがある．そこで，プロセスの実行状態を 1nd OS から 2nd OS に引き継ぎ，アップデート後もプロセスを継続実行する必要がある．

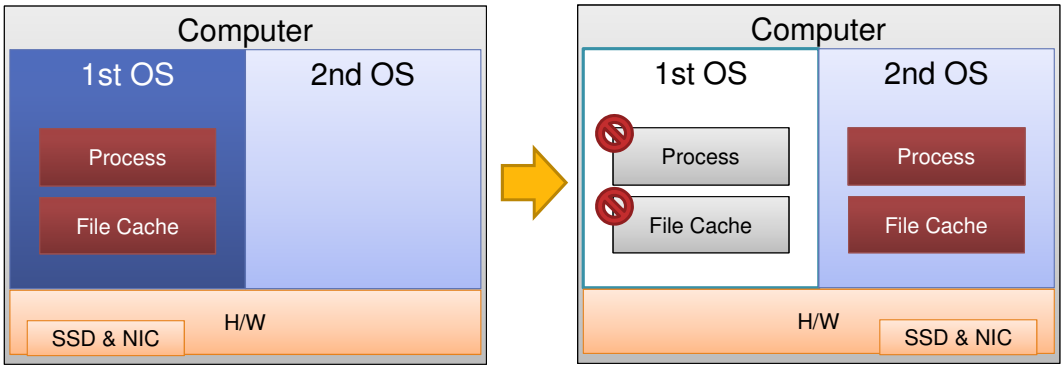


図 2　アップデート直前とアップデート直後の様子

3.2　OS 間でのデータ移行

前節で示した要件を満たすために解決すべき問題は，OS カーネル依存のデータ構造を異なる OS バージョン間で移行する方法である．例えば，ファイルキャッシュとプロセスのマイグレーションは，1st OS の内部状態を 2nd OS に移す処理であると言える．ここで，1st OS と 2nd OS が同バージョンかつ同コンフィギュレーションであればデータ構造に差異が無いため，内部状態の移行にデータ構造の変換処理は必要ない．しかし，アップデート実施時は 1st OS と 2nd OS のバージョンが異なる．OS バージョンが異なれば，カーネル内部のデータ構造も異なる可能性が高いため，内部状態の以降にはデータ構造の変換が必要である．

Seamless Kernel Update[5] では，アップデート前の OS カーネル上でユーザレイヤにおける活動から得られるデータ，すなわちシステムコールやネットワークを通じて得られたデータをチェックポイントとして保存し，アップデート先の OS でチェックポイントに基づいてアップデート前のファイルやソケットの状態を再現する．この方法により，高レイヤーの情報を用いることで OS カーネルの内部構造を隠蔽したまま，異なるバージョン間でデータの移行を行っている．しかし，OS カーネル固有のデータによる情報の補完が必要であり，高レイヤーの情報のみでは，内部的なデータ構造を完全に再現できていない．

そこで，OS カーネルバージョンに起因するデータ構造の差異を吸収し，OS カーネルバージョンに依存しない内部状態の移行を実現する．OS バージョンに起因するデータ構造の差異の吸収方法として，コンパイラを用いる方法 [6] や，構造体のメンバ変数のオフセットを用いる方法がある．

コンパイラを用いる方法では，データ構造をビルド時に解析し，既存のデータ構造とは別に，共通書式のデータ構造を定義する．アップデートプログラムがこのデータ構造を元にアップデート前のデータを解析することで，OS 固有のデータ構造への依存性を無くしている．しかし，コンパイラの改良が必要な上，場合によっては OS カーネル自体をそのコンパイラに対応させる実装が求められるため，実装コストが高いと言える．

オフセットを用いる方法では，構造体の先頭アドレスと移行対象となるメンバ変数の構造体内のオフセット値の和によって構造体メンバを参照し，差異を吸収する．更新前のデータ構造の全容を更新後の OS が把握する必要はなく，構造体の先頭アドレスと対象メンバのオフセット値に基づいて参照ができる．しかし，移行対象となるメンバ変数に関するオフセット値を求める必要があることから，部分的な差異の吸収が必要な場合に適している．

本手法では吸収すべきデータ構造はカーネルのデータ構造の一部であることから，オフセットを用いたデータ構造の参照方法を採用した．2nd OS から 1st OS のメモリ領域を解析し内部構造を移行する際に，構造体のメンバを辿るが，このとき，メンバ名で参照するのではなく，事前に取得した 1st OS 固有のオフセット値を用いることでデータ構造を吸収する．

3.3　適用可能範囲

本手法は，1st OS と 2nd OS のデータ構造の差異を構造体内のオフセットによって表現することで吸収し，データ構造の変化に対応する手法である．しかし，データ管理方法の差異には対応できない．例えば，現在の Linux Kernel では，ファイルキャッシュを radix-tree と呼ばれる木構造を用いて管理しているが，Linux Kernel 2.6.5 以前では単純なリンクリストによって管理されていた [7]．この場合，Kernel 2.6.5 以前の OS カーネルからそれ以降の OS へアップデートすることは出来ない．

また，データの管理方法自体に変更は無くとも，構造体のメンバ名が変更される場合や，データ移行に必要なメンバ変数がバージョンアップで無くなる場合，別の構造体に移動する場合にはオフセットでは表現できないため，表現方法を今後検討する．

4.　設計と実装

多重 OS 機構 Orthros をベースにバージョンの異なる Linux カーネル（2.6.38.7，3.0.1.1，3.4.75）に対して提案手法の実装を行った．

4.1 実装概要

提案手法の必要要件を満たすために，Orthros の各機構を基盤に採用する．デバイスマイグレーションは Orthros の機構をそのまま使用することができる．なぜなら，1st OS と 2nd OS はデバイスを排他利用しており，アップデート時に内部情報を移行させるのではなく，単純に 1st OS が使用していたデバイスを 2nd OS が初期化して使用するためである．

次に，Orthros の死活監視機構を，1st OS の停止を知らせる停止通知機構として使用する．しかし，停止通知機構では 1st OS が停止したことを通知することが目的であるため，Orthros と異なり生存通知用の IPI を定期的に送信する必要はない．そこで，1st OS が停止した時にのみ IPI を 2nd OS に対して送信するように改良を加える．

ファイルキャッシュマイグレーションとプロセスマイグレーションについては，OS カーネルのバージョン依存のデータ構造を 1st OS と 2nd OS で移行するため，オフセットを用いてデータ構造の差異を吸収する改良が必要となる．現在までに，ファイルキャッシュマイグレーション機構に対する提案手法の実装のみが完了しているため，以降では，このファイルキャッシュマイグレーション機構に焦点を当てる．なお，プロセスマイグレーションについても今後対応するが，2nd OS から 1st OS のメモリに残留したデータ構造を解析することに変わりはないため，同様の方法で実現できると考えている．

4.2 ファイルキャッシュマイグレーションの改良

Orthros のファイルキャッシュマイグレーション機構を基盤に，OS バージョンに依存しないファイルキャッシュマイグレーション機構を実装した．本節では，オフセット情報を用いた OS カーネル依存のデータ構造の差異吸収を実現する実装について述べる．

4.2.1 オフセット情報の共通定義

Orthros のファイルキャッシュマイグレーション機構の実装から，ファイルキャッシュマイグレーションに必要となる構造体と，そのメンバ変数を調査した．その結果，それらのメンバ変数のオフセット値を格納する構造体を，オフセット構造体として定義した．**図 3** に，提案手法におけるオフセット構造体の定義を示す．オフセット構造体は，2nd OS で動作するマイグレーション機構に依存するのみであり，実装対象の 3 バージョンのカーネルで共通である．バージョンによらず同一の構造体を宣言することで，どの OS バージョン間でもオフセット構造体のメンバ変数へ正確にアクセスすることができる．オフセット構造体は，マイグレーションに用いる構造体と一対一で対応させ，参照する構造体の型名に接頭辞 “offset_” を付加したものを型名とした．また，メンバの名前は，参照するメンバ変数名と同一とした．例えば，super_block 構造体のメンバ変数を参照する場合には，offset_super_block 構造体のメンバ変数に格納されるオフセット値を用いる．

```
struct offset_super_block {
  memaddr_t s_bdev;
  memaddr_t s_bdi;
  memaddr_t s_instances;
};

struct offset_inode {
  memaddr_t i_sb;
  memaddr_t i_wb_list;
  memaddr_t i_ino;
  memaddr_t i_size;
  memaddr_t i_mapping;
};

struct offset_address_space {
  memaddr_t host;
  memaddr_t page_tree;
  memaddr_t nrpages;
  memaddr_t backing_dev_info;
};

struct offset_backing_dev_info {
  memaddr_t capabilities;
  memaddr_t wb;
};

struct offset_block_device {
  memaddr_t bd_inode;
  memaddr_t bd_disk;
};

struct offset_page {
  memaddr_t flags;
  memaddr_t index;
  memaddr_t _count;
  memaddr_t virtual;
};
```

図 3 オフセット構造体の定義

```
struct offset_super_block offset_sb {
  .s_bdev = offsetof(struct super_block, s_bdev);
  .s_bdi = offsetof(struct super_block, s_bdi);
  .s_instances = offsetof(struct super_block, s_instances);
};
```

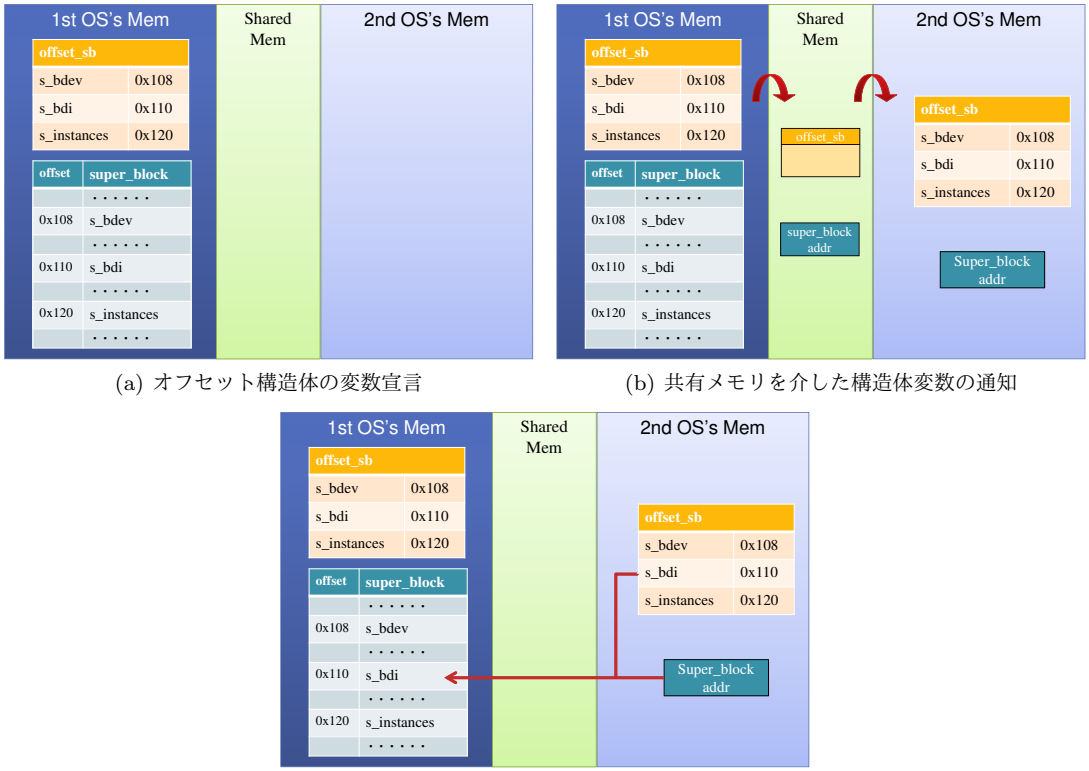
図 4 オフセット構造体の変数宣言例

なお，連結リストを管理する list_head 構造体もマイグレーションに用いられていたが，各バージョン通して変化はなく，今後も構成が変わることは考えづらいため，オフセット構造体の定義から除外した．

現在は，必要となる構造体やメンバ変数を手作業により抽出しているが，今後は効率的な抽出方法の検討を進める予定である．

4.2.2 オフセット構造体の変数宣言

オフセットを用いて 2nd OS が 1st OS のメモリ領域を解析するには，1st OS 上でオフセット構造体の変数を宣言



(a) オフセット構造体の変数宣言

(b) 共有メモリを介した構造体変数の通知

(c) 異バージョン OS 固有のデータ構造の参照

図 5 オフセット構造体の使われ方

表 1 それぞれのバージョンの super_block のメンバ変数のオフセット値

	Kernel 2.6.38.7	Kernel 3.0.1.1	Kernel 3.4.75
s_bdev	0x108	0x108	0x160
s_bdi	0x110	0x110	0x168
s_instances	0x120	0x120	0x178

し，2nd OS に対し通知する必要がある．

ここで，構造体のメンバ変数のオフセット値はカーネル動作中に変わることはないため，カーネル固有の静的な数値であるといえる．Linux カーネルには，構造体の型と，その構造体に所属するメンバ変数名を引数としてとる offsetof() マクロが存在する．このマクロを使用すると，カーネルビルド時に引数に与えたメンバ変数のオフセット値が計算され，そのオフセット値に置き換えられる．

本手法では，この offsetof() マクロを用いて，対応するオフセット値をセットしたオフセット構造体の変数をカーネル内に宣言した．**図 4** は offset_super_block 構造体の変数宣言の例である．提案手法では，図 3 に示したすべてのオフセット構造体に対して，図 4 と同様に構造体変数を宣言した．これにより，カーネル内でオフセット構造体を利用できる．**図 5(a)** の offset_sb が 1st OS 起動後にメモリ上に存在するオフセット構造体の変数にあたる．

表 1 に実装対象の 3 バージョンの Linux Kernel における，super_block 構造体の解析で用いるメンバ変数の実際のオフセット値を示す．これらの値は，それぞれのカーネルでオフセット構造体のメンバ変数の値を printk() 関数により出力することで取得した．Kernel 2.6.38.7 と Kernel 3.0.1.1 ではオフセット値に変化はないが，これは super_block 構造体の構成がほぼ変わっていないことを示している．実際に，Kernel 2.6.38.7 と Kernel 3.0.1.1 の super_block 構造体の宣言を確認したところ，Kernel 3.0.1.1 の super_block 構造体に s_uuid メンバと，cleancache_poolid メンバが追加されていたことが分かった．しかし，この 2 つのメンバ変数は，s_instances メンバ以降に宣言されているため，本オフセット値に変化はなかった．一方，Kernel 3.4.75 では他の 2 つと比較してオフセット値が大きく異なっている．これは，super_block 構造体に変化があったためである．Kernel 3.0.1.1 と Kernel 3.4.75 の super_block 構造体を比較すると，s_bdev メンバよりも前方に複数のメンバが追加されていることを確認した．なお，これらの値は，本実装に使用した計算機環境である Arch Linux 上でビルドしたカスタムカーネルの固有値であるため，コンフィグレーションや環境によっては，異なる場合がある．

4.2.3 構造体変数の事前通知

2nd OS ではオフセット構造体に含まれるオフセット値と構造体の先頭アドレスに基づいて 1st OS のメモリを解析する．このため，1st OS 上で宣言されたオフセット構造体の変数を，1st OS から 2nd OS に対して事前に通知する必要がある．1st OS がオフセット構造体の変数を，2nd OS の起動直後に 1nd OS と 2nd OS の間に設けられた共

表 2　評価環境

	1st OS	2nd OS
CPU コア	3	0,1,2
メモリ	576-960MB	0-576MB,960-8192MB
Device	SSD,NIC(動作確認用)	SSD,NIC

表 3　アップデートパターン

	1st OS	2nd OS
パターン A	Linux Kernel 3.0.1.1	Linux Kernel 3.4.75
パターン B	Linux Kernel 2.6.38.7	Linux Kernel 3.4.75
パターン C	Linux Kernel 2.6.38.7	Linux Kernel 3.0.1.1

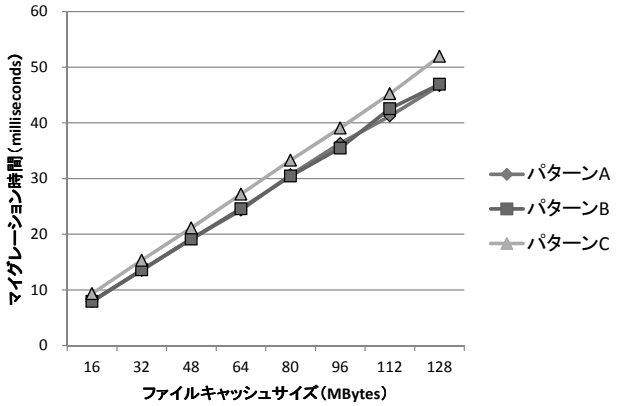


図 6　ファイルキャッシュマイグレーション機構の評価結果

有メモリに書き込むことで通知する（図 **5(b)** 参照）．この時，ファイルキャッシュマイグレーションの対象となるファイルシステムの super_block 構造体のアドレスも書き込む．super_block 構造体を通じて辿ることで，他のデータにアクセスできるためである．

4.2.4　構造体メンバの参照方法の変更

Orthros を基盤とした本提案手法のマイグレーション機構に対して，構造体のメンバ変数に対する参照方法を，メンバ変数名からオフセット値による参照方法へ変更した．図 **5(c)** は 1st OS における super_block 構造体の s_bdi メンバを 2nd OS から参照する例を示している．C 言語の仕様上，ポインタに対する加減算は通常の数値計算と挙動が異なり，そのポインタが指す型によって結果が異なる．例えば，int 型のポインタ変数に対して 1 を加算した場合，ポインタ変数の指すアドレス値は int 型のサイズ分加算されることになる．そのため，1st OS の構造体のアドレスを整数値として扱うことで，2nd OS では単純な整数演算を用いてアドレス値が計算でき，1st OS のデータ構造の解析が可能となる．また，2nd OS によるデータ構造の解析には Linux 既存の関数を多く使用するが，データ移行に関わる構造体を参照する関数に関しては，オフセットによる参照方法に変更した関数を別途定義した．提案手法で別途定義した関数は 6 つで，実装コストはわずかである．

5.　評価

Linux Kernel のバージョン 2.6.38.7，3.0.1.1，3.4.75 の 3 つに対して提案手法を実装し評価を行った．評価環境を**表 2** に示す．評価として，**表 3** に示す 3 つのアップデートパターンで，2nd OS におけるファイルキャッシュのマイグレーションに要する時間を，1st OS に残存したファイルキャッシュのサイズを変えて計測を行った．

まず，すべてのアップデートパターンについて，正常にアップデートできたことを確認した．次に，評価結果を図 **6** に示す．どのパターンにおいてもキャッシュのサイズに比例して処理時間が増加していることが分かる．16MB のファイルキャッシュのマイグレーション時で約 8msec，128MB のファイルキャシュマイグレーション時は約 50msec で完了する．ギガバイトオーダーのキャッシュを書き込むことは稀であり，ファイルキャッシュがある程度大きくなっても高速に動作し，処理時間も無視できるほどの時間だと言える．

パターン C（バージョン 2.6.38.7 から 3.0.1.1）におけるマイグレーションが，他の 2 パターンと比較して低速である．マイグレーション先が Kernel 3.4.75 であるパターン A，パターン B の場合、ほぼ同等の結果が出ていることから，Kernel 3.0.1.1 にこの原因があると考えられるが，原因の特定には至っていない．ただ、マイグレーション処理はどのバージョンでも大きな差異はないため，カーネルのコンフィギュレーションの違いが影響していると推測している．今後，コンフィギュレーションを変更したカーネルで実験を行い，この原因を明らかにしたい．

また，提案システムはプロセスマイグレーションに未対応であるが，参考までに現実装における 1st OS から 2nd OS へのアップデートに要する時間を計測した．2nd OS が 1st OS からの停止通知を受けた瞬間から，デバイスマイグレーション，ファイルキャッシュマイグレーション，NIC に対する IP アドレスの割り当ての終了までを，移行に要する時間とした．結果として，約 690msec〜約 740msec で完了した．なお，処理時間の幅はマイグレーションするファイルキャッシュのサイズに依存するものである．この処理時間からファイルキャッシュマイグレーションの処理時間を除いた時間は約 680msec で，デバイスマイグレーションに多くの時間を費やしていることが分かった．今後，プロセスマイグレーションを実現して評価を行う予定であるが，プロセスマイグレーションは，ファイルキャッシュと同様に 1st OS のメモリ領域の解析と内部状態を 2nd OS に移す処理である．このため，プロセスマイグレーションに対応した場合でも，処理時間の増加を少量に抑えられると予想している．この点から，現実装におけるアップデート処理時間は十分期待できる値であると言える．

6. 関連研究

本章では，単一計算機内での OS の多重起動に関する研究と，OS のアップデートに関連する研究について述べる．

Otherworld[8] は，OS をマイクロリブートすることによってプロセスの実行状態を保護する手法である．OS に障害が発生した際に，新たな OS をウォームブートすることで，停止した OS のメモリイメージを残存させる．この残存しているプロセスに関するデータを用いて，新しい OS はプロセスの復元し，実行を再開する．メモリイメージの参照によりデータを復元するという点では Orthros と同様であるが，フェイルオーバー時にカーネルをウォームブートする処理が，ダウンタイムの短縮を妨げていると言える．

Seamless Kernel Update[5] はチェックポイントを用いて，プロセスの実行状態を保持したままカーネルをアップデートする手法である．この手法では，チェックポイント用スレッドが動作するプロセッサ以外を一時停止することで他のスレッドを停止させ，メモリの一貫性を保証し，メモリに記録されたデータ構造をチェックポイントとしてメモリに保存する．そして，Otherworld と同様に新たなカーネルをウォームブートし，保存したチェックポイントを復元することで，実行状態を失うことの無い OS カーネルのアップデートを実現している．この手法は，カーネルからユーザレイヤでの活動に基づいてアップデートできることを目指しているが，不足している情報に関してはシステム開発者による修正を行っている．一方，提案手法では，データ構造の差異をオフセット構造体として定義することで，複数のバージョン間でのデータ移行に対応している．

Ksplice[9] は Oracle Linux の機能として提供されている，動作中のカーネルに対し動的にパッチを当てる手法である．パッチ適用前のカーネル（pre）とパッチ適用後のカーネル（post）を比較し，更新された関数群を動作中のカーネルコード領域へ配置し，修正対象の関数実行時に更新後の関数へ jmp 命令を発行することで，柔軟なカーネルのライブアップデートを実現している．Ksplice はソースコードを独自のオブジェクト形式に変換する必要があるため，環境に大きく依存するが，本手法では Linux カーネルに直接手を加えているため，従来の Linux カーネルとしても使用でき，環境への依存度が低い．

7. まとめ

本稿では，カーネルアップデートに係るダウンタイムの短縮を目指し，多重 OS 機構を用いたカーネルのライブアップデート手法を提案した．フェイルオーバーを実現する多重 OS 機構 Orthros を基盤に，OS バージョンに依存した内部構造の差異を吸収するオフセット構造体を導入し，プロセス状態を引き継ぐ OS カーネルのライブアップデートを実現するシステムについて述べた．開発のプロトタイプとして，Orthros に実装されたファイルキャッシュマイグレーション機構を，オフセットを用いたデータ構造の参照方法に変更することで，OS バージョンに依存しない機構に改良した．この結果，メモリ上のファイルキャッシュを，異なるバージョンの OS 間で保護することに成功した．提案手法を評価した結果，128MB のファイルキャシュのマイグレーションに約 50msec を要し，現実装でのシステム全体の復帰までに約 700msec を要した．提案手法のうち，OS バージョンに依存しないプロセスマイグレーション機構は実装が完了していないため，早急に実装し，評価を行う予定である．

参考文献

[1] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pp. 305–318 (2011).

[2] Hariprasad, N.: Kexec を使って Linux の起動を早める，IBM（オンライン），入手先 ⟨https://www.ibm.com/developerworks/jp/linux/library/l-kexec/⟩（参照 2014-06-03）.

[3] 吉田健二，齋藤彰一，毛利公一，松尾啓志：プロセスの耐障害性向上のための多重 OS の開発と評価，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 7, No. 2, pp. 11–24 (2014).

[4] Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pp. 355–364 (2008).

[5] Siniavine, M. and Goel, A.: Seamless kernel updates, *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pp. 1–12 (2013).

[6] Giuffrida, C. and Tanenbaum, A.: Safe and automated state transfer for secure and reliable live update, *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*, pp. 16–20 (2012).

[7] 高橋浩和，小田逸郎，山幡為佐久：Linux カーネル 2.6 解読室，ソフトバンククリエイティブ (2006).

[8] Depoutovitch, A. and Stumm, M.: Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes, *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 181–194 (2010).

[9] Arnold, J. and Kaashoek, M. F.: Ksplice: Automatic Rebootless Kernel Updates, *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pp. 187–198 (2009).