Vol.2014-OS-130 No.24
2014/7/29

プロセスコンテキストを維持したオペレーティングシステムのアップデート手法

寺田 献[1,a)]　山田 浩史[1,b)]

概要：オペレーティングシステム（OS）のアップデートの適用するために行う再起動は，OS 上で動作するプロセスのプロセスコンテキストを失ってしまう．本研究では OS アップデートのための再起動を行いつつも，プロセスコンテキストを維持する手法について提案する．OS は対象となるプロセスを停止させ，プロセスコンテキストをアップデートによって変化しない *Shelter Object* に格納する．Shelter Object をハイパーバイザ上に保存して OS を再起動し，再起動後に再び Shelter Object を受け取ってプロセスコンテキストを復元する．維持可能となったコンテキストは，CPU の実行状態，メモリ内容，ファイル，パイプ，シグナルである．提案手法を Xen 4.2.1 および Linux 2.6.32.61 に実装した．評価実験の結果，400MB のヒープメモリを持つプロセスは，最も高速な実装方法では退避に 4ms，復元に 67ms かかることがわかった．この時間は通常は OS 再起動の時間に比べてはるかに短く，このシステムによるダウンタイムは無視できる程度である．

1. はじめに

オペレーティングシステム（OS）カーネルの信頼性を向上するために，アップデートを迅速に行うことが推奨されている．アップデート内容には，脆弱性のもととなるバグの修正，プログラムの最適化に伴うコードの変更，新しい機能の追加などが挙げられる．これらのアップデートは OS カーネルの内部を変更するため，変数やデータ構造，関数などの位置が変化する可能性があり，これらを利用するコードにも修正が必要である．このような理由から，OS カーネルに対するアップデートの適用は再起動をすることが一般的である．

OS カーネルの再起動時には，その OS カーネル機能に依存しているプロセス全てを終了する必要がある．このときプロセスの実行状態（コンテキスト）は失われてしまう．たとえば，サーバプログラムではメモリ上のデータキャッシュであり，ワードプロセッサソフトであればリドゥ機能のための履歴などである．これらのデータを失うということは，性能の劣化や機能の低下を意味する．コンテキストを不揮発性のディスク装置へ保存するという手法もあるが，プロセス終了時の書き込みとプロセス再開時の読み込みに時間がかかり，サービスのダウンタイムは更に長くなる．このようなプロセスコンテキストの喪失は，プロセスを継続して使用したいユーザとしては避けたいことである．

本研究では，プロセスコンテキストを維持しつつ，OS カーネルをアップデート可能にする手法の提案を行う．プロセス主体のコンテキスト保存ではなく，OS カーネル主体で CPU，メモリ，I/O に関連した情報を保存する．これらの情報を保持するデータ構造から，プロセスコンテキストの維持に必須のものを選び，退避用データ構造の *Shelter Object* に格納する．格納が終わった後，ハイパーバイザ上に Shelter Object を移動する．OS カーネルを再起動した後，Shelter Object をハイパーバイザから受け取り，この情報を元にプロセスコンテキストを復元する．Shelter Object 内の情報は，OS カーネル内のデータ構造に直接代入するか，システムコールに対応する関数などの引数として利用する．この手法によって，OS カーネルのアップデートを行う際にも，プロセスが提供するサービスのダウンタイムを減らしつつ，性能劣化を抑止する．

実装にはハイパーバイザに Xen 4.2.1，OS に Linux 2.6.32.61 を用いた．OS 上で動作するプロセスの CPU のレジスタ情報，メモリ内容，ファイルの展開情報，シグナルハンドラ，パイプの利用情報を，プロセスの退避・復元処理の前後で維持しつつ OS カーネルを再起動することに成功した．ユーザメモリの大きさが 400MB 程度のプログラムであれば，退避に 4ms，復元に 67ms と，本研究によるダウンタイムは OS 再起動の時間に比べはるかに短かった．

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[a)] terada@asg.cs.tuat.ac.jp
[b)] yamada@asg.cs.tuat.ac.jp

本研究報告では，第 2 章で関連研究の抱える問題点を示し，第 3 章でそれらを解決する手法を提案する．第 4 章で，目的を達成するシステムの設計を述べ，第 5 章で Linux と Xen を用いた実装例について示す．本研究で作成したプログラムの評価は第 6 章で行う．第 7 章では研究目標と本研究の達成内容を比較し，考察を行う．

2. 関連研究

本章では，本研究に関連した研究について紹介し，その問題点を指摘する．

2.1 OS カーネルのホットアップデート

OS カーネルのアップデートの適用を，OS カーネルの再起動無しに行う手法．DynAMOS[1] と Ksplice[2] では，アップデート内容は OS カーネルモジュールとして用意し，アップデートのタイミングで適用対象の関数への CALL 命令を OS カーネルモジュールへのジャンプコードに書き換えることで，処理を切り替えている．OS カーネルに専用のコードを用意しておく必要はないが，アップデート内容に合わせた独自のパッチを必要とする．Ksplice ではこのパッチを自動で生成することができる．DynAMOS ではアップデートパッチの分類を行い，適切なアップデート適用タイミングについて議論しており，多くのものが自動決定できることを示している．この決定には，アップデート適用対象の関数・データ構造が現在参照されていない，参照を行うプログラムがスタック上にない，といったことが関与する．ホットアップデートでは，必ずしもすべての種類のアップデートに対応したパッチを作成できるわけではない．たとえばデータ構造への追加や，複数箇所を同時に修正する場合などである．本研究ではアップデート内容に依存しないシステムを目指す．

2.2 OS カーネルの高速再起動

OS カーネルの再起動によるサービスのダウンタイムを減らすために，再起動の所要時間を減らすための工夫が行われている．ShadowReboot[3] では，仮想マシンモニタを用いて運用中の OS が動作している仮想マシン（VM）とは別の VM を用意し，その VM 上でアップデートを適用した OS カーネルを起動する．初期化処理が終了しログイン画面となった瞬間に元の VM 上に OS カーネルのコンテキストを上書きし，アップデートを適用した OS カーネルの実行に移行する．ダウンタイムは OS カーネルのコンテキストを上書きする時間とログイン後のユーザレベルのプログラムの初期化が中心であり，OS カーネルの再起動にかかる時間は通常の再起動に比べ 90%以上削減できている．BIOS を介したハードウェアの初期化を省略する Kexec という手法が存在し，この仕組を利用した kboot[4] はブートローダの代わりとなって新しい OS カーネルを起動することができる．これらは OS のアップデートが高速に適用できる反面，プロセスのコンテキストは失われてしまう．Phase-based Reboot[5] はリブート処理を段階ごとに分け，それぞれの処理で得られた結果を保存し，再起動時には可能な限り再利用している．この手法ではメモリイメージを再利用するため，OS カーネルにアップデートを適用することはできない．

2.3 プロセスマイグレーション

プロセスを別のマシン上で動作している OS カーネルに移動する．再起動対象となっている OS 上で稼働中のプロセスを別の OS 上に移動し，再起動を行うことでプロセスの実行を継続したまま OS のアップデートを行うことができる．Zap[6] では，プロセスと OS カーネルのレイヤの間に，プロセス ID やファイルディスクリプタ番号などを仮想化するレイヤを挿入することで，OS カーネル内での ID のコンフリクトを避けつつ，プロセスから見える ID は変化させない，といったことを可能にしている．移送するプロセスは予めこの仮想化レイヤの上で動作させておかなければならないといった制約の他，プロセスが利用しているメモリ量が増えればデータ転送量も増え，ダウンタイムとネットワークなどのデバイス負荷を増大させてしまう．

3. 提案

3.1 アップデートとコンテキスト維持の分離

OS カーネルのアップデートと，プロセスコンテキストの維持に要する処理は独立させる．プロセスコンテキストはアップデートの前後で変化しない Shelter Object に格納する．OS カーネルのアップデートによっては，構造体にメンバが追加されたり，関数や命令の追加によって隣接するシンボルの位置が変化する可能性がある．カーネル内のプロセスコンテキストのうちデータ構造となっているものは，必要なメンバを選定した上で新しく新しく Shelter Object 用のデータ構造を定義する必要がある．またメンバのうちカーネル空間のデータ構造を指すポインタとなっているものは，指している先のデータ構造を別途保存する必要がある．

3.2 プロセスコンテキストの選定

プロセスコンテキストは OS カーネル内のデータ構造で管理されている．実行状態や割り込み待ちの停止状態などのフラグや，仮想メモリアドレスと物理メモリアドレスの対応関係であるページテーブル，展開しているファイルのパスや展開フラグ，読み書きの位置などが挙げられる．これらは OS の設計や実装に依存するものであるが，類似・共通する機能を表 1 のように分類することができる．回復すべきプロセスコンテキストは主にこれらの情報である．これらに関連するデータ構造や，システムコールに必要とな

表 1 各資源から選出したコンテキスト

資源	内容
CPU	レジスタの値，プロセス ID（親子・兄弟関係，グループ）
メモリ	メモリ領域の使用状況
I/O	ディスクリプタ ID と読み書きの位置，バッファ
通知	シグナルの受信状態と対応関数の登録情報

る引数などの情報を元に，各 OS に対して Shelter Object に格納する情報を選定する．

3.3 提案手法の長所

アップデートは通常の再起動を経て適用されるため，どのような場合にも OS カーネル自体に完全なアップデートを行うことができる．OS カーネル内のデータ構造の変化があったとしても，Shelter Object は変わらないためデータの受け渡しはアップデートの前後で正しく行われる．アップデート後の OS は変化したデータ構造も正しく参照することができるため，Shelter Object のデータから正しい位置に値を設定できる．既存のプロセスには一切変更を加える必要がない．プロセスから参照可能な資源は，再起動の前後で同じものを提供することを目指しているため，プロセスに対して再起動に対処する関数の作成を要求することはない．使用する資源は仮想マシン 1 台であり，特別なハードウェアや別のマシンを必要としない．またプロセスマイグレーションのように，プロセスが利用する資源を仮想化するために，追加の仮想化レイヤなども必要としないため，平常時のオーバヘッドはハイパーバイザによるもののみである．

3.4 適切な退避・復元手法

プロセスコンテキストを取得するために，プロセスがユーザ空間で停止している状態を作り出す．カーネル空間で停止している場合はユーザ空間に戻るまで待つか，システムコールに割り込みをかけるなどして一度ユーザ空間に戻すべきである．これは命令レジスタには現在実行中のカーネルコードのアドレスが入っているため，アップデート後のカーネルコードのレイアウトが変化した場合に誤った位置から命令を実行してしまうことを防ぐためである．このような理由から，カーネルのコードのみを実行するカーネルスレッドは，本手法でプロセスコンテキストを維持することができない可能性がある．作成した Shelter Object は，OS の再起動によって失われることのない場所に格納する．またその場所へのアクセス速度は高速であることが望ましい．

4. 設計

本章では，適切な Shelter Object を定義手法と，プロセスコンテキストを退避・復元する手法の概要を述べる．

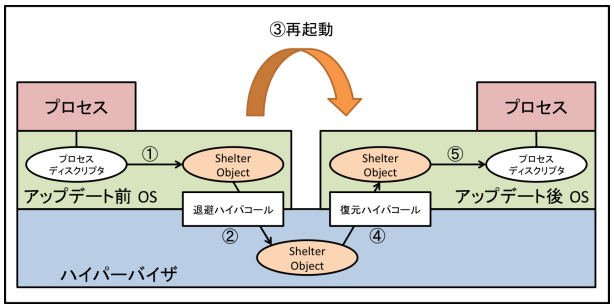

図 1 プロセスコンテキストを維持したアップデート手法

4.1 Linux に対する Shelter Object の設計

Linux では主に，表 2 に挙げる構造体によってプロセスを管理している．task_struct, thread_struct, sighand_struct, mm_struct，files_struct は 1 つのプロセスに対して 1 つ，それ以外はコンテキストに合わせて変化する．これらのプロセスコンテキストを格納できるように Shelter Object を設計した．Shelter Object は主に，プロセスごとに存在する固定長データを格納する shelter_t 構造体，可変長となる sig_shelter_t 構造体，vma_shelter_t 構造体，pte_shelter_t 構造体，file_shelter_t 構造体，xpid_t 型の配列で構成される．表 2 に各データ型・データ構造に格納する情報を示す．

4.2 プロセスコンテキストの退避・復元手順

プロセスコンテキストの退避・復元の手法を図 1 に示す．OS カーネル内にあるプロセスディスクリプタのうち，①選定したプロセスコンテキストを OS が Shelter Object に格納し，②退避ハイパーコールによってハイパーバイザに保存してプロセスを停止させる．③OS を再起動してアップデートを適用し，④復元ハイパーコールでハイパーバイザから退避していた Shelter Object を受け取り，⑤その情報を元にプロセスを再構築する．プロセス ID など，コンフリクトが懸念される資源は，該当資源の初期化時に使用フラグを立ててコンフリクトを避ける．

5. 実装

5.1 開発環境

ハイパーバイザに Xen 4.2.1，OS に Linux 2.6.32.61 を使用した．Linux は準仮想化されており，CPU の一部のレジスタの操作などのハードウェアへの特権的な命令，ページテーブルの更新などはハイパーコールを通して行われる．OS の起動オプションは，ASLR（仮想アドレスのランダム化）の機能は使用しない，シングルユーザモードで起動する，とした，ハイパーバイザに 328 行，ゲスト OS に 611 行のコードを追加した．Intel x86 アーキテクチャのうち，64 ビットで動作するものを対象とした．

5.2 各資源の退避と復元処理

CPU のレジスタ情報は，Linux がコンテキストスイッ

表 2　Linux での Shelter Object を構成する主要なデータ構造

構造体	内容	対応する Linux データ構造
shelter_t	固定長データ構造すべて，可変長配列の長さとアドレス	task_struct，thread_struct，mm_struct
sig_shelter_t	シグナルハンドラの情報	sighand_struct，k_sigaction
vma_shelter_t	メモリ領域の情報	vm_area_struct
pte_shelter_t	ページの使用状況に関する情報	pte_t，page
file_shelter_t	展開したファイルに関する情報	file
pipe_shelter_t	展開したパイプに関する情報	pipe_inode_info
xpid_t	子プロセス，兄弟プロセスのリスト	pid_t

チを行うためにレジスタの値を格納する thread_struct 構造体を定めている．この値は直接レジスタに代入されることから，退避時に Shelter Object にはこの値を直接コピーし，復元時にも代入すれば良い．

ユーザ空間のメモリ内容の退避にはいくつかの実装方法があり，詳細は次節で述べる．メモリの利用状況については，vm_area_struct のリストをたどりながら，仮想アドレスとページのフラグを取得する．この仮想アドレスをページテーブルで探索してページテーブルエントリを取得する．またページフレームに対応する page 構造体から，ページの利用情報を取得する．

ファイルとパイプは fd_array を通して取得する．fd_array は file 構造体の配列を持っており，その添字はファイルディスクリプタ番号である．file 構造体にはそのファイルを操作するための操作関数をまとめた file_operations 構造体へのポインタをメンバに持っており，このポインタにあるデータ構造によって，通常のファイルであるか，パイプの読み込み側，パイプの書き込み側，を区別することができる．通常のファイルであれば，file 構造体からパスと読み書きなどの属性を保存する．ファイルは閉じる時にディスク装置に書き込まれるため，ファイルの中身を Shelter Object 内に保存する必要はない．ファイルの復元時にはパスを元に OS がファイルを展開する関数を呼び出し，lseek() 関数によってファイルの読み込み位置を設定する．パイプであれば，file に対応する inode を取得し，パイプ内のデータを read() 関数によって Shelter Object に保存する必要がある．復元時には新しいパイプを作成して，write() 関数によってパイプに書き込む．これらの資源に対して正しいファイルディスクリプタ番号を割り当てるために，通常の展開関数の一部を変更し，指定したファイルディスクリプタ番号でファイルやパイプを展開できる関数を作成した．

シグナルに対しては，プロセスが独自の処理関数を sigaction() システムコールで登録することができる．関数ポインタはユーザ空間の仮想アドレスを保持するため，この値は変更すること無く再利用することができる．その他，システムコールで必要となる引数はカーネル内の k_sigaction 構造体内に入っているため，退避時にはこれらの値を Shelter Object に格納し，復元時にはこれらの値を引数として sigaction() システムコールに対応する関数を OS カーネルから実行する．

5.3　メモリ内容の復元手法

カーネル内のデータ構造とは異なり，ユーザ空間内のメモリ内容は必須のものであるかの判定を OS から行うことはできない．ただし，書き込み禁止のメモリ領域は実行ファイルなどからプロセスの起動時にロードされたものであり，これを退避しなかったとしても復元することは可能である．そのためメモリ内容の退避が必須となるのは書き込み可能なページである．ページ内容はプロセスによっては膨大であるため，可能な限り複製するべきではない．次章では，複製を行う回数を減らすことでプロセスコンテキストの退避と復元にかかる時間を評価する．ページ内容を維持するために，３種類の方法で実装を行った．以下にその特徴を示す．

①アップデート前の OS からハイパーバイザに複製し，ハイパーバイザからアップデート後の OS に複製する（複製２回）

②アップデート前の OS のページをハイパーバイザが参照し，ハイパーバイザからアップデート後の OS に複製する（複製１回）

③アップデート前の OS のページをハイパーバイザがアップデート後の OS のページテーブルに割り当てる（複製０回）

複製の回数が減ればより高速に退避・復元ができることが期待できるが，複製回数を減らすためには一般的により多く OS を変更する必要がある．複製２回の実装はどの OS でも利用できるものであるが，複製１回の実装や複製０回の実装は必ずしも利用できるとは限らない．

5.3.1　複製２回の実装

ページごとにハイパーバイザ内にページサイズ分のメモリを動的に確保する．その領域にハイパーバイザがメモリ内容をコピーすることで，ページの複製を作成する．OS の再起動後には，OS がプロセスのメモリページを確保した後に，その領域にハイパーバイザからコピーをする．この実装はアップデート前の OS が元のページにアクセスを続けることができるため，OS によっては適した実装方法

である．

5.3.2 複製１回の実装

ページの所有権をハイパーバイザに譲渡する．それ以降アップデート前の OS はそのページにアクセスすることは許されない．アクセスを行う OS であれば，その部分を修正する必要がある．コピーオンライトなどで書き込み可能なページが共有されていることがあり，その場合にはアップデート前の OS のページテーブルからページの参照を消すなどして，

アップデート後の OS がプロセスにページを割り当てた後，その領域にメモリ内容をコピーする．

5.3.3 複製０回の実装

アップデート後の OS に対してもページテーブルの変更で参照するページを変える．ページ内容のコピーは一切発生しないため，高速に動作することが予想される．OS の実装によっては，複数のページテーブルから同じページを参照する可能性があり，この場合には全てのページテーブルを書き換える必要がある．Linux にはページテーブルの逆方向である，メモリページから仮想アドレスへのマッピング（逆マッピング）を扱うデータ構造が存在するが，そのようなデータ構造がない場合には複製１回の実装か複製２回の実装を利用するべきである．

6. 評価

6.1 実験環境

開発環境と同じ環境において評価を行った．CPU は Intel Xeon E3-1270 V2（動作周波数 3.50GHz，4 コア，8 スレッド，キャッシュ 8MB），メモリは 10GB である．仮想マシンの構成は，仮想 CPU1 コア，仮想メモリ 4GB，スワップ領域を作成しない，ネットワークデバイスを利用しない，とした．

6.2 実験の目的

作成したプログラムは，CPU，メモリ，ファイル，パイプ，シグナルについての情報を，OS の再起動の前後で全て再現することができた．ここでは退避・復元の処理性能を評価するために，退避するプロセスのメモリ容量を変化させたときの処理時間の変化と，メモリ内容の複製方法を変化させたときの処理時間の変化を調査する．プロセスコンテキストの退避と復元にかかる時間はサービスのダウンタイムとなるため，極力短いことが好ましい．時間の計測対象からファイルやパイプを除外した理由は，予備実験においてユーザ空間内のメモリ情報を維持するためにかかる時間は，これらの資源を維持するためにかかる時間より多く時間を消費するという結果が出たためである．

6.3 メモリサイズによる退避・復元の処理時間の変化

ファイルやパイプを展開せず，10 ページ（40KB），1,000 ページ（4MB），100,000 ページ（400MB）分のヒープ領域を動的に確保する評価用プログラムを用いて，退避処理と復元処理にかかる時間を測定した．処理はいくつかのフェーズに区切り，それぞれのフェーズで時間を計測した．メモリ内容の維持には複製１回の実装で評価した．退避処理の所要時間を図 2，復元処理の所要時間を図 3 に示す．

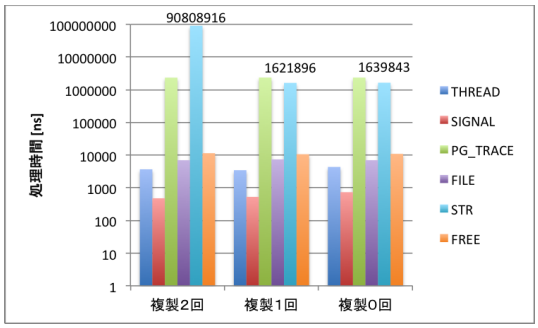

図 2 ヒープページ数による退避処理時間の変化

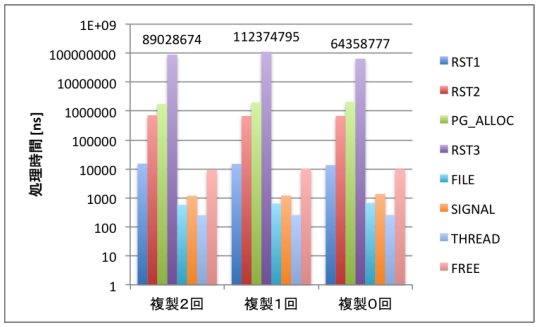

図 3 ヒープページ数による復元処理時間の変化

図 2 の各フェーズで行う処理は，THREAD は CPU の実行状態の保存，SIGNAL はシグナルハンドラの保存，PG_TRACE は利用しているメモリページ情報の保存，FILE はファイルとパイプの保存，STR はハイパーバイザ内への Shelter Object の保存，FREE は Shelter Object 作成のために確保した一時的なメモリ領域の解放，である．

利用しているページ数に比例して PG_TRACE と STR の処理時間が増えている．STR 内ではページ内容の複製は行われていない．それ以外の値にはほとんど変化がない．

図 3 の各フェーズで行う処理は，RST1 は Shelter Object のうち固定長のデータ構造の受け取りハイパーコール，RST2 は可変長のデータ構造の受け取り，PG_ALLOC は利用しているページフレームに実メモリを確実に割り当てる処理，RST3 はページ内容の受け取り，FILE はファイルとパイプの復元，SIGNAL はシグナルハンドラの再設定，THREAD は CPU の実行状態の復元，FREE は Shelter Object 受け取りのために確保した一時的なメモリ領域の解放，である．

利用しているページ数に比例して，RST3 が極めて長い時間を要している．これはページ内容の複製が行われるためである．ページ数が増えるに従って Shelter Object の

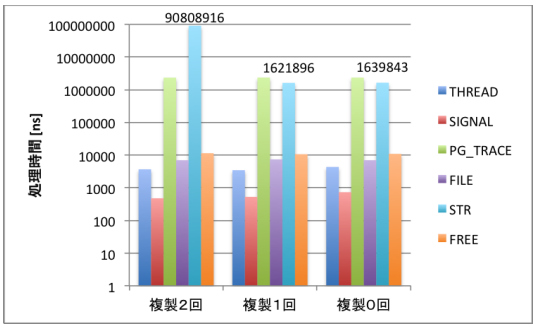

図 4 実装方法よる退避処理時間の変化

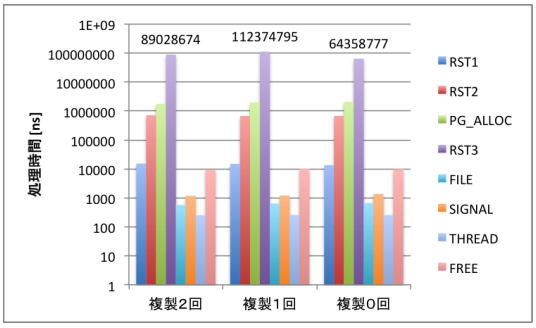

図 5 実装方法よる復元処理時間の変化

pte_shelter_t 構造体を受け取る量も増えるため，RST2 も所要時間が増加している．

6.4 ページ内容維持の実装方法による変化

３種類の実装方法の比較を行った．ヒープメモリのページ数は 100,000 ページとし，ファイルやパイプは展開していない．フェーズの区切りは前の実験と同じである．

退避処理のうち変化したのは STR であり，ページ内容の複製から参照をハイパーバイザに渡す手法に変えることで，約 56 倍の高速化を達成することができた．それ以外は変更が無いため，処理時間も変化していない．

復元処理では RST3 の所要時間が，複製２回と複製１回とで同じ程度の時間がかかっている．複製１回の実装の方が約 20ms ほど時間が長くかかる理由は，RST3 の時点で参照を保持しているページを仮想アドレスに割り当てるためにページテーブルを変更し，それからメモリ内容の複製を行っているためである．そのため複製２回の実装よりもページテーブルを変更する時間だけ処理時間が長くなる．複製０回の実装は複製２回に比べて約 1.4 倍の高速化にとどまっている．複製０回の実装がそれほど高速にならない理由は２つ考えられる．１つは，ページテーブルの変更に要する時間が複製０回の実装でもかかってしまうことである．もう１つは，CPU 上のメモリキャッシュや TLB キャッシュのフラッシュに時間がかかってしまうことである．

複製０回の実装は複製２回の実装に比べて，退避・復元ともに高速化できた．表 3 に，各実装方法，各ヒープページ数での退避・復元処理の合計時間を示す．

表 3 退避・復元処理の合計時間 [μ s]

実装方法	処理	10 ページ	1000 ページ	100,000 ページ
複製２回	退避	38	986	93,174
	復元	37	940	91,542
複製１回	退避	42	96	4,002
	復元	48	1,106	115,075
複製０回	退避	30	85	4,015
	復元	80	785	67,160

7. 結論

本研究ではプロセスコンテキストを維持した，OS のアップデート手法を提案した．本手法は以下の三要素によって成り立つ．1.Shelter Object によって OS のアップデート内容とプロセスコンテキストに関する情報を切り離す．2.OS カーネルから取得した復元に必要なプロセスコンテキストを Shelter Object に格納し，これをハイパーコールでハイパーバイザ上に保存する．3.OS カーネルの再起動が完了した後に，ユーザは任意のタイミングで退避したプロセスを復元する．本研究ではケーススタディとして Linux と Xen を用いたが，Shelter Object は特定の OS，ハイパーバイザに依存することなく作成することができ，Linux 以外の OS に対しても本研究の手法を適用することができる．維持することが可能であった資源は，CPU，メモリ，ファイル，パイプ，シグナルであり，既存のユーザプロセスに対して変更を加える必要はない．

7.1 今後の課題

7.1.1 ダウンタイムの短縮

プロセスの退避・復元にかかる時間は，通常十秒程度以上のオーダーがかかるブート時間に比べ，十分に短いことがわかった．今後は退避・復元の時間を短縮するのではなく，アップデート後の OS のブートにかかる時間を短縮する．本研究ではハイパーバイザを用いたのは，一つの物理マシン上で複数の OS を実行することができるためでもある．

アップデート適用のためのリブート時間を短縮する手法は以下の通りである．アップデート前の OS は起動しつつ，アップデート後の OS を別の仮想マシン上で起動する．アップデート後の OS の初期化が終了してユーザプロセスが起動する前に一時停止させる．アップデート前の OS はプロセスの実行を止め，退避するプロセスに対して Shelter Object を作成し，ハイパーバイザに退避する．ハイパーバイザは仮想アドレスの OS 領域が，アップデート後の OS のメモリ上のイメージ指すようにページテーブルを変更する．本手法と同様の手法で，アップデート後の OS で Shelter Object からプロセスコンテキストを再構築する．

7.1.2 退避・復元対象資源の拡大

プロセス間データ通信であるソケットはパイプと同様に接続情報とバッファ内容とに分けて Shelter Object に格納することで，復元ができるものと思われる．スワップ領域は，OS カーネルの初期化時に利用状況も初期化されるが，その直後にスワップ領域の利用状況を Shelter Object から挿入することで，アップデート後の OS がメモリ内容のある位置に上書きをすることを防ぐことができる．現在プロセス ID は Shelter Object 内に格納してはいるものの，任意のプロセス ID でプロセスを生成する関数が無いために復元できていない．fork() と execve() に似た挙動をする関数を作成し，同じプロセス ID でプロセスコンテキストを再開させる．

参考文献

[1] Makris, K. and Ryu, K. D.: Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels, *EuroSys '07 Proceedings of the 2nd ACM European Conference on Computer Systems*, pp. 327–340 (2007).

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

[3] Yamada, H. and Kono, K.: Traveling Forward in Time to Newer Operating Systems using ShadowReboot, *Proceeding VEE '13 Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 121–130 (2013).

[4] Almesberger, W.: kboot - A Boot Loader Based on Kexec, *Proceedings of the Linux Symposium, 2006, Ottawa, Canada*, Vol. One, pp. 27–38 (2006).

[5] Yamakita, K., Yamada, H. and Kono, K.: Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery, *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 169–180 (2011).

[6] Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments, *ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation Homepage archive Volume 36 Issue SI, Winter 2002*, pp. 361–376 (2002).