Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

プログラム領域のcopy on writeを抑制した複数プロセスの動的更新手法の提案

小澤 駿[1,a)]　齋藤 彰一[1]

概要：増加する脆弱性に対応するために，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必要がある．この状況に対してソフトウェアの可用性を損ねない更新手法として，ソフトウェアを止めることなく更新する手法である動的更新手法が提案されている．しかし，従来の動的更新手法は単一のプロセスの更新を対象としており，同一バイナリの複数プロセスを更新するにはプロセス数に比例した時間だけプロセスを停止させなくてはならない．本稿ではプロセスの停止時間削減を目的とした動的更新手法を提案する．プログラムコード領域に対する copy on write の実行を無効化することで，親プロセスへのメモリ書き込みを子プロセスに反映させ，同一バイナリの複数プロセスの更新を同時に行う．これにより従来の動的更新手法に比べ，プロセスの動的更新による停止時間を削減する．

キーワード：動的アップデート，停止時間削減，copy on write，マルチプロセス

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イルス感染，外部からの不正アクセスの原因となる．そのためソフトウェアは定期的なアップデートによって，脆弱性を含まない最新の状態に保つことが求められている．

通常，ソフトウェアのアップデートでは，最新のバージョンのインストールとソフトウェアの再起動という 2 つの処理が必要である．このためアップデート前のソフトウェアの実行停止からアップデート後のソフトウェアの実行再開まで，アップデート対象となるソフトウェアの動作は停止する．このソフトウェアのダウンタイムは，ソフトウェアの運用における可用性の低下させる．また再起動により実行停止直前のソフトウェアの実行状態が失われるため，起動直後のソフトウェアは初期状態から処理が始まる．Web サーバやデータベースサーバといった多数にサービスを提供するシステムをアップデートする場合，アップデートはサービスの継続運用に問題を発生させ，サービス品質を低下させる．

アップデートにおけるソフトウェアのダウンタイムを削減する手法として動的アップデート手法 [1] がある．動的アップデート手法では，アップデート対象となるソフトウェアを停止することなくアップデートすることを目的としている．動的アップデートでは，メモリ上の実行バイナリを書き換えることで，アップデート対象となるプロセスを停止することなくバイナリを修正する．

しかし，既存の動的アップデート手法は単一のプロセスをアップデート対象としている．そのため，Apache[2] に代表される Web サーバのように，親プロセスが大量の子プロセスを生成するアプリケーションのアップデート，つまり同一バイナリの複数プロセスのアップデートには適さない．このような場合には，生成された各子プロセスに対して個別にアップデート処理を行う必要があり，子プロセス数に比例した時間だけプロセスを停止させなくてはならない．したがって，多くの子プロセスを生成して処理を行う Web サーバやメールサーバの動的アップデートには，アップデート対象プロセスに対して大きな停止時間が必要となる．

本稿では，同一バイナリの複数プロセスをアップデートする際の，アップデート対象プロセスの停止時間削減を目的とした動的アップデート手法を提案する．同一バイナリの複数プロセスにおけるメモリ管理機能 copy on write を抑制し，親プロセスへのメモリ書き込みを子プロセスのメモリにも反映させることで，複数プロセスのアップデートを同時に行う．これにより，同一バイナリの複数プロセスに対する動的アップデートが発生させるダウンタイムを削減し，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．

[1] 名古屋工業大学
Nagoya Institute of Technology
[a)] liveupdate@mail.ssn.nitech.ac.jp

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

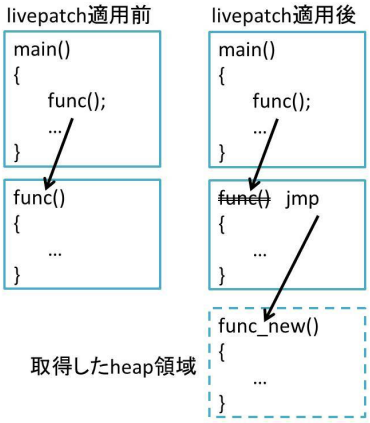

図 1　jmp 命令を用いた関数単位での動的アップデート手法

また，動的アップデートによるソフトウェアのアップデートを実行ファイルに反映させる機能も提案する．この機能によって，ソフトウェアは実行停止後もアップデート状態を維持する．

本稿は，2 章で関連する動的アップデート手法に関する研究について述べる．次に，3 章では提案手法の概要を述べ，4 章で提案手法実現のための実装について述べる．5 章では提案手法の評価を述べ，最後に 6 章でまとめる．

2. 関連研究

本章では，ソフトウェアとカーネルに対する動的アップデートに関連した研究について述べる．なお，本稿では，関数単位の動的アップデートについて，アップデート対象となるプロセスを更新対象プロセス，アップデート対象となる関数を更新対象関数，アップデート後に実行される関数を更新済関数と言う．

2.1 livepatch

livepatch[3] はユーザランドソフトウェアの動的アップデートを目的としたソフトウェアである．ptrace システムコールを利用することで，更新対象関数を含むプロセスのメモリ空間を書き換え，動的アップデートを行う．**図 1** は，livepatch を用いた動的アップデートによって，更新対象関数 `func` を更新済関数 `func_new` にアップデートした場合の関数の呼び出し関係を示している．livepatch は，ptrace システムコールを用いて実行した mmap2 システムコールで確保した heap 領域に，更新済関数 `func_new` の命令列を書き込む．その後，関数 `func` の先頭を，関数 `func_new` への jmp 命令に置き換える．これにより，関数 `func` が実行されると，jmp 命令を経て，代わりに関数 `func_new` が実行される．以上により関数単位の動的アップデートを実現する．

2.2 Ksplice

Ksplice[4] は Oracle Linux[5] の一部として提供される，動作中のカーネルに対して動的にパッチを適用する機能である．Ksplice は，まずソースコードの修正パッチを用いて，アップデート後のバージョンのカーネルを作成する．そして，このアップデート後のカーネルとアップデート前のカーネルのバイナリを比較することで，更新対象関数の検知と，更新済関数を取得する．その後，Ksplice は更新済関数をカーネルコード領域へ配置し，更新対象関数の先頭に更新済関数の先頭への jmp 命令を挿入することで，関数単位の動的アップデートを実現する．

2.3 pannus

pannus[6] は，mmap2 システムコールを改良することで，ユーザプロセスと Linux Kernel の動的アップデートを実現している．pannus も livepatch と同様に，heap 領域を確保して更新済関数を配置し，更新対象関数の先頭に jmp 命令を配置する．livepatch が heap 領域を確保するには，ptrace を利用して更新対象プロセスに mmap2 システムコールを実行させる必要がある．一方，pannus では，mmap2 システムコールを改良して特定のプロセスのメモリ管理構造体にアクセスできるシステムコールを追加実装することで，更新対象プロセスの heap 領域を直接確保している．また，マルチスレッドで処理を行うプロセスの動的アップデートに対応している．

2.4 kaho

kaho[7] は，更新対象プロセスの停止時間削減を目的とした動的アップデート手法である．jmp 命令の代わりに breakpoint による割り込みを用いている．また，ptrace システムコールによるメモリ書き込みの停止時間を，プログラム・カウンタのチェック機能によって削減している．これらの実装により，従来の動的アップデート手法より更新対象プロセスの停止時間を削減している．

2.5 POLUS

POLUS[8] は，マルチスレッドで処理を行うサーバプロセスをアップデート対象とした動的アップデート手法である．ptrace システムコールを利用することで動的ライブラリを更新対象プロセスに取得させ，更新済関数を実行領域に挿入している．

2.6 関連研究のまとめ

関連研究では，いずれも単一プロセスをアップデート対象として想定している．そのため関連研究を用いて Web サーバのように親プロセスが大量の子プロセスを生成する場合，つまり同一バイナリの複数プロセスを動的アップデートする場合，生成された子プロセスの数に比例した長

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

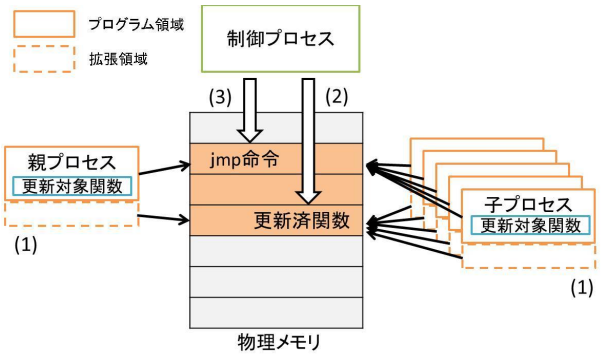

図 2　提案手法の概要

さの停止時間が必要となる問題がある．

3. 提案

本稿では，既存の動的アップデート手法が想定しなかった，同一バイナリの複数プロセスに対する動的アップデート手法を提案し，更新対象プロセスの停止時間を削減する．本提案手法は，同一バイナリの複数プロセスがメモリ管理に用いる機能である copy on write の動作を変更することで，親プロセスのアップデートを子プロセスにも反映させる．これによって，既存の動的アップデート手法ではプロセスごとに必要であったアップデート処理が，提案手法では同時に複数プロセスに対して適用でき，アップデート処理に伴う更新対象プロセスの停止時間を削減できる．また，動的アップデート手法の可用性拡大のために，更新対象プロセスの実行ファイル書き換え機能を実現した．この機能により，動的アップデートの更新対象プロセスの再起動後もアップデート内容を維持できる．

本章では，提案手法の概要，更新済関数を配置するプログラム領域のセクション (以下，拡張領域という) の取得，メモリ管理機能 copy on write の概要とその抑制手法について述べる．さらに実行ファイル書き換え機能について述べる．

3.1 提案手法の概要

提案手法は関数単位の動的アップデートを行う．提案手法において，動的アップデート処理を主に行うプロセスを制御プロセスと呼ぶ．提案手法における動的アップデートの処理を以下に示す．また，**図 2** に提案手法の概要を示し，図中の番号は以下の処理手順の番号を指す．

(1) アップデートの準備として，更新対象プロセスは，自身の実行前に更新済関数用の拡張領域を確保する．
(2) アップデート処理は，制御プロセスが拡張領域に更新済関数を書き込むことで行う．この際，拡張領域に対する copy on write が動作しないようにする．
(3) 制御プロセスは，更新対象関数の先頭に拡張領域へのjmp 命令を書き込む．この際，拡張領域と同様に，更新対象関数の先頭を含むページの copy on write が動作しないようにする．

(1) で取得する拡張領域は更新済関数を格納するためのメモリ空間である．本手法ではこの拡張領域の取得は動的に行わず，更新対象プロセスの実行前に行う．(2) と (3) は，制御プロセスが更新対象プロセスのメモリ空間へ書き込みを行う．この際に copy on write を動作させないことで，親プロセスだけでなく子プロセスにも書き込みを反映させる．

3.2 拡張領域の取得

提案手法は，更新対象プロセスに拡張領域を確保する．既存の動的アップデート手法である livepatch では，ptrace システムコールを用いることで更新対象プロセスに mmap2 システムコールを呼び出させ，動的にメモリ空間を確保させる．この方法の場合，確保できるメモリ空間は heap 領域となるため，プロセス間で共有できない．提案手法では，プロセス間で共有しているメモリ空間内に拡張領域を確保するため，更新対象プロセスの実行前に，静的にプログラム領域に追加のセクションを取得する．

3.3 copy on write

本節では，メモリ管理機能 copy on write の概要について説明し，提案手法における利用方法について述べる．

3.3.1 概要

copy on write は，子プロセス生成時には親プロセスのメモリ空間を子プロセスのメモリ空間に複製せず，メモリ空間に書き込みが行われた時に初めてプロセス固有のメモリ空間を取得する仕組みである．つまり，メモリ書き込みが行われるまでは，親子プロセスは同じメモリ空間を所有している．

メモリ書き込みが行われた場合，Linux Kernel は書き込み対象となった物理メモリページとは別の物理メモリページを取得する．次に，新しく取得した物理メモリページに，当該メモリページの内容をコピーした後でメモリ書き込みを行う．その後，書き込みを行ったプロセスの当該仮想メモリページの物理メモリページとの関連付けを，新たに取得した物理メモリページに変更する．以上の処理により，共有しているメモリ空間は変更されず，書き込みを行ったプロセスにだけ書き込み内容が反映される．**図 3** は，copy on write が有効になった状態でのメモリ書き込みの動作を示している．左側が親プロセスの仮想メモリ空間を，右側が子プロセスの仮想メモリ空間を，中央が物理メモリを示しており，ページ単位で区切られている．各仮想メモリページは，対応する物理メモリページと関連付けられている．図 3 の場合，親プロセスの上から 3 つ目の仮想メモリページに書き込み (図 3 の (1)) が行われた際に copy on write が動作し，書き込み前まで指していた物理メモリペー

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

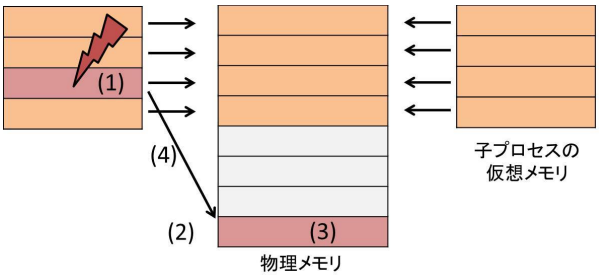

図 3　copy on write が有効になった状態でのメモリ書き込みの動作

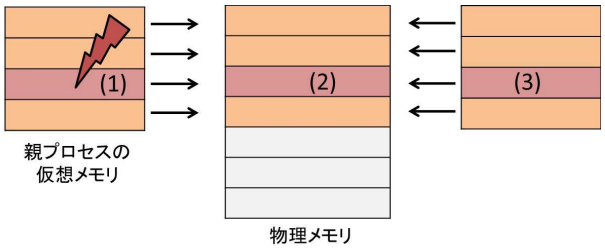

図 4　copy on write を抑制した状態でのメモリ書き込みの動作

ジとは別の物理メモリページを確保 (図 3 の (2)) して，その物理メモリページに書き込み (図 3 の (3)) を行う様子を示している．その後，書き込みが行われた仮想メモリページのリンクを新しく取得した物理メモリページに張り替える (図 3 の (4))．

3.3.2　copy on write の抑制手法

提案手法は，拡張領域に更新済関数の書き込む時と，更新対象関数の先頭に jmp 命令を書き込む時に，これらのページに対する copy on write を無効にする．copy on write が有効な状態では，書き込まれた内容はプロセス固有のメモリ空間に複製されて他のプロセスのメモリ空間は影響を受けない．しかし copy on write を無効にすることで，すべてのプロセスが共有しているメモリ空間を書き換えることができ，親子プロセスのメモリ空間を同時に書き換えることが可能となる．**図 4** は copy on write が抑制された状態でのメモリ書き込みの動作を示している．親プロセスの仮想メモリ空間に書き込み (図 4 の (1)) が行われた際，仮想メモリページが現在指している物理メモリページに書き込み (図 4 の (2)) を行う．この物理メモリページは子プロセスと共有しているため，子プロセスの仮想メモリページも同様に書き込み (図 4 の (3)) が反映される．

3.4　実行ファイル書き換え機能

提案手法は，動的アップデートにおける書き込み内容を，当該プロセスの実行ファイルに反映する．既存の動的アップデート手法では，ptrace システムコールや heap 領域といった動的に確保できる領域を利用してアップデートを実現している．そのため，アップデートしたプロセスが終了して再度起動した場合，先に実施したアップデート内容は失われているため，再度動的アップデートを行う必要

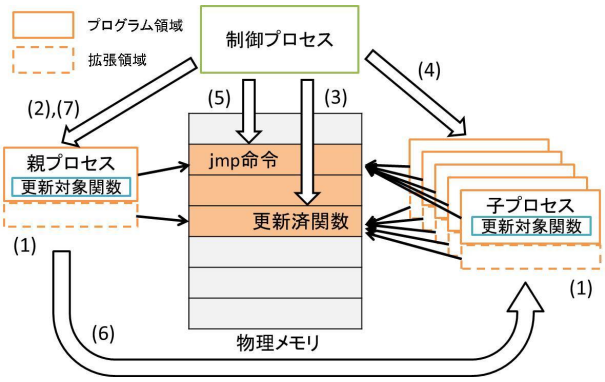

図 5　提案手法による動的アップデートの概要

がある．本提案では，更新対象プロセスの実行ファイルをアップデート内容で書き換えることで，再起動時にもアップデート状態を維持できる．

アップデート時の内容は，ファイルキャッシュに書き込まれている．このファイルキャッシュは，既存の動的アップデート手法では破棄されるが，提案手法では通常のファイルキャッシュ同様に書き戻しを行う．また，この実行ファイル書き換え機能の実行を実際に実行するか否かは，ユーザが選択できる．提案手法では，更新対象プロセスの実行ファイルのバックアップを動的アップデート前に作成する．ユーザが実行ファイルの書き換えを望まない場合，再実行時にバックアップのバイナリを使用することで，アップデートが行われる前のプロセスを起動させることができる．

4.　実装

提案手法を Linux Kernel 3.14.4 に実装した．また提案手法を利用した制御プロセスを実装した．本章では，制御プロセスの実装の詳細について述べる．

4.1　制御プロセスの動作概要

提案手法を実装した制御プロセスにおける基本的な動的アップデートでは，livepatch 同様に，ptrace システムコールを用いて更新済関数の書き込みと jmp 命令の書き込みを行う．**図 5** に提案手法による制御プロセスの概要を示す．まず，アップデートの準備段階として，更新対象プロセスは拡張領域を確保 (図 5 の (1)) する．更新対象プロセスが起動した後，制御プロセスは `PTRACE_ATTACH` リクエストを親プロセスに対して行い，動作を中断 (図 5 の (2)) させる．その後，制御プロセスは拡張領域に対する copy on write の動作を抑制するための無効化と，更新済関数の書き込み (図 5 の (3)) を行う．次に，子プロセスのプログラム・カウンタ (以下，PC という) を調査 (図 5 の (4)) する．これは，制御プロセスの jmp 命令の書き込みによる，子プロセスの不正な実行を回避するためである．この調査の後，更新対象関数の先頭に jmp 命令を書き込む (図 5 の

(5))．また，子プロセスの PC の調査によって，いくつかの子プロセスの実行が中断する場合がある．そのため，親プロセスが，実行を中断した子プロセスの復帰処理 (図 5 の (6)) を行う．最後に，制御プロセスは親プロセスの処理を再開 (図 5 の (7)) させ，動的アップデートを終了する．その後，アップデート内容を更新対象プロセスの実行ファイルに保存する．以下，各処理について述べる．

4.2 拡張領域の取得

更新対象プロセスに更新済関数を配置するための拡張領域の取得は静的に行う．実装には，objcopy コマンドを利用した．objcopy コマンドを利用することで，プログラム領域と同じパーミッションを持つ別の領域を，更新対象プロセス内に割り当てることができる．プログラム領域と同じパーミッションを持たせることで，ptrace システムコール以外の方法による書き込みを禁じ，命令の実行が可能となる．

4.2.1 拡張領域における問題点

mprotect システムコールはプロセスのメモリ領域のパーミッションを変更するシステムコールである．mprotect システムコールを用いて，拡張領域のパーミッションを書き換え可能に変更することで，プロセス内で拡張領域に対する自己書き換えが可能となり，悪意ある命令が実行できる．

4.2.2 拡張領域に対するパーミッション変更抑制機構

拡張領域の安全性を確保するために mprotect システムコールを拡張し，拡張領域を対象範囲とした実行については，root 権限以外での mprotect システムコールを失敗させる実装を行った．拡張後の mprotect システムコールは，ELF ヘッダから拡張領域の有無を確認し，拡張領域を含む場合はそのアドレス範囲を取得する．さらに，実際に mprotect システムコールが root 権限以外で実行された場合には，要求されたアドレス範囲と拡張領域のアドレス範囲を比較し，拡張領域に対する要求の場合はエラーを返す．

4.3 copy on write の抑制

copy on write を抑制するために，copy on write 処理の無効化処理 (以下，CoW 無効化処理という) と CoW 無効化対象のプロセスとアドレス範囲を登録するシステムコールを実装した．copy on write の実行を判断する処理は Linux Kernel 内の `do_wp_page` 関数内で定義されている．提案手法では，この `do_wp_page` 関数内に CoW 無効化処理を追加した．CoW 無効化処理では，copy on write の処理対象となるプロセスが CoW 無効化対象のプロセスであり，かつ書き込み範囲が CoW 無効化対象のアドレス範囲である場合，copy on write を行わずに共有しているページに書き込みを行う処理に移行する．この処理により，動的アップデート時に書き込みを行うプログラム領域と拡張領域では copy on write が実行されず，親子プロセスが同時にアップデートできる．

CoW 無効化対象のプロセスとアドレス範囲の登録は，新規にシステムコールを実装することで実現した．この CoW 無効化対象登録システムコールは，CoW 無効化対象のプロセス，開始アドレスとサイズを表す変数に登録する．これらのシステムコールを，アップデート処理実行前に呼び出すことで，更新済関数等の書き込みが親子プロセスで共有される．

4.4 子プロセスの不正な実行について

提案手法における，子プロセスのアップデートに関する問題点について述べる．通常，ptrace システムコールを用いてメモリ書き込みを行う場合，対象となるプロセスに対して `PTRACE_ATTACH` リクエストを行うことで一時的に動作を中断させ，メモリ書き換えを行う．これに対し，提案手法では親プロセスのメモリを書き換えることで，子プロセスのメモリも書き換える．ptrace システムコールによるメモリ書き込み処理の直接の対象は親プロセスのみであるため，`PTRACE_ATTACH` リクエストは親プロセスの実行のみを中断させる．一方，子プロセスは `PTRACE_ATTACH` リクエストを受けていないため，実行は中断されない．そのため，子プロセスが実行している最中の命令を ptrace システムコールで書き換える可能性がある．この場合，不正な命令が実行され，子プロセスの処理が異常停止する可能性がある．この問題を解決するためには，ptrace システムコールで書き換えを行う前に，子プロセスが不正な命令を実行する可能性がある状態か，確認するための機能が必要となる．

以下本節では，4.4.1 で，jmp 命令書き込み時の子プロセスの不正な実行について述べ，4.4.2 以降でその解決方法について述べる．

4.4.1 jmp 命令を書き込む際の問題点

更新対象関数の先頭に jmp 命令を書き込む際の問題について述べる．提案手法では動的アップデートのために，5 バイトの jmp 命令をアップデート対象となる関数の先頭に書き込む必要がある．しかし，例えば，子プロセスが関数の先頭で実行されるベースポインタの push 命令 (1 バイト) を実行しているときに jmp 命令の書き込みが行われると，子プロセスは jmp 命令の 2 バイト目から実行しようとする．jmp 命令の 2 バイト目から 5 バイト目は jmp 先のアドレスであるため，子プロセスの実行によって異常終了することが考えられる．つまり，子プロセスの PC が指すアドレスが，jmp 命令の書き込み範囲であった場合，不正な実行が行われる．**図 6** は，更新対象関数の先頭に jmp 命令を書き込む前後の更新対象関数の様子を示している．jmp 命令書き込み前は，PC が mov 命令の先頭アドレスを指していたが， jmp 命令書き込み後は，jmp 命令の 2 バイト目を指している．

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

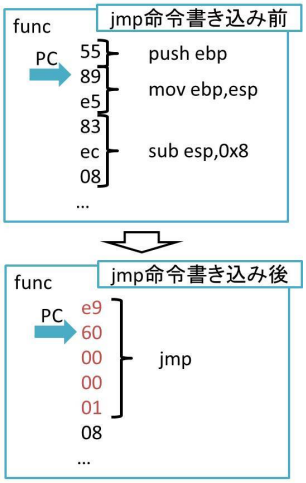

図 6　更新対象関数の先頭に jmp 命令を書き込んだ際の問題

4.4.2　問題点の解決方法

すべての子プロセスの PC が jmp 命令の書き込み範囲の外であるときに jmp 命令が書き込まれた場合には，子プロセスの不正な命令の実行は行われない．そのため，jmp 命令の書き込み範囲を異常終了誘発区間とし，また，すべての子プロセスの PC が異常終了誘発区間以外を指している状態を安全な状態と定義する．そして，すべての子プロセスが安全な状態であることを確かめた後，jmp 命令を書き込むことで，子プロセスの不正な実行を回避する．すべての子プロセスが安全な状態か確認するためには，以下の 2 つを行う必要がある．

- 子プロセスの PC が異常終了誘発区間にあるかを調べる．
- 新たな子プロセスの異常終了誘発区間の実行を防ぐ．

この 2 つについて，4.4.3 と 4.4.4 で述べる．また 4.4.5 で，子プロセスの PC が異常終了誘発区間を指していた場合の，制御プロセスの動作について述べる．

4.4.3　子プロセスの異常終了誘発区間の実行有無の判断

子プロセスの PC を調べるために，PC を含むタスクの状態を管理している `task_struct` 構造体を調べるシステムコール (以下，PC 調査システムコールという) を実装した．子プロセスの `task_struct` 構造体は，親プロセスの `task_struct` 構造体の sibling メンバを先頭に環状リストで管理されている．そのためシステムコールの引数に親プロセスの process id を渡すことですべての子プロセスの PC が異常終了誘発区間を指しているか否かを判断できる．

4.4.4　新たな子プロセスの異常終了誘発区間の実行防止

PC 調査システムコールの実行時間は，子プロセスの増加に比例して長くなる．そのため，PC 調査システムコール開始時には異常終了誘発区間を実行していなかった子プロセスが，PC 調査システムコール終了時には異常終了誘発区間を実行している可能性がある．この問題を解決するために，新たな子プロセスが異常終了誘発区間内の命令の実行をすることを防ぐ機能が必要となる．

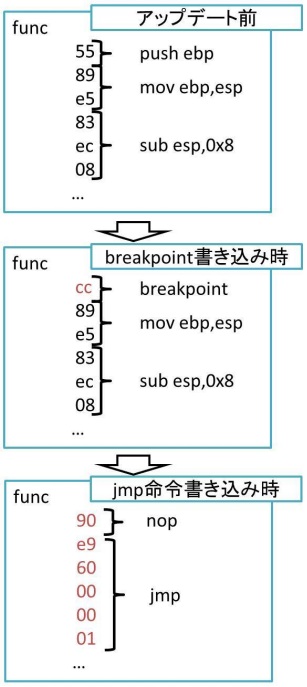

図 7　更新対象関数の命令変更遷移

提案手法では，PC 調査システムコールの実行前に更新対象関数の先頭に breakpoint を書き込む．この breakpoint により，新たに異常終了誘発区間を実行しようとする子プロセスは停止状態となる．**図 7** に，breakpoint 挿入を含む，更新対象関数の変更手順を示す．図は 3 つの更新対象関数の状態を示している．上は，動的アップデートを行う前の更新対象関数を示している．中央は，breakpoint の書き込み時の更新対象関数を示しており，下は PC 調査システムコール実行後に，jmp 命令が書き込まれた際の更新対象関数を示している．breakpoint は，x86 アーキテクチャでは 1 バイトで表現されるため，jmp 命令書き込み時のような問題は生じない．なお，breakpoint で停止した子プロセスの PC は更新対象関数の先頭から 1 バイト先を示すため，PC が異常終了誘発区間を示すことになる．これに対して，PC 調査システムコールは，`task_struct` 構造体から breakpoint を実行して停止状態か実行状態かを，state メンバを参照することで判断できる．したがって，breakpoint によって停止中のプロセスを，異常終了誘発区間を実行中と誤判定することはない．最後に，jmp 命令書き込みにおいて，breakpoint を nop 命令に置き換えて，jmp 命令を更新対象関数の先頭の 1 バイト先から書き込む．これにより，breakpoint を実行した子プロセスが再スケジューリングされたときに，PC を変更することなしにアップデート後の関数に jmp する．

4.4.5　子プロセスが異常終了誘発区間にいた場合の動作

PC 調査システムコールによって，PC が異常終了誘発区間を指す子プロセスが一つでも存在することを検知した場

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

合，制御プロセスはこの子プロセスが異常終了誘発区間の実行終了まで停止後，再度異常終了誘発区間を実行していた子プロセスの PC を調べる．この制御プロセスの停止時間中は，アップデート対象の親子プロセスは通常の動作を継続する．

4.5 Breakpoint の実行と復帰処理

親プロセスが breakpoint を実行した場合，制御プロセスは親プロセスの PC を更新済関数の先頭に変更し処理を再開させる．これによって親プロセスの停止時間を削減する．

一方，子プロセスが breakpoint を実行した場合は処理が中断する．処理が中断した子プロセスの復帰処理は，親プロセスが ptrace システムコールで子プロセスに SIGCONT シグナルを送ることで行う．実行が中断した子プロセスは，親プロセスから送られてくる SIGCONT シグナルを受け取ることで，実行を再開する．

子プロセスの復帰処理を親プロセスが実行するには，制御プロセスが以下の処理を行う．

(1) 拡張領域に復帰処理を書き込む．
(2) 現在の親プロセスのレジスタの値を取得し，保存する．
(3) 親プロセスの PC を復帰処理の先頭アドレスに変更する．
(4) 復帰処理処理の実行終了を検知する．復帰処理の最後に breakpoint を配置することで，親プロセスは復帰処理終了時に breakpoint を実行する．制御プロセスは，親プロセスの breakpoint 実行を検知できるので，復帰処理の終了を検知できる．
(5) 親プロセスのレジスタを，保存していた復帰処理実行前のレジスタの値で復元する．

4.6 実行ファイルの書き換え

提案手法では，動的アップデートにおけるアップデート内容を，実行ファイルに保存する．この実行ファイルの書き換え機能の実装は Linux Kernel における書き戻し処理を拡張することで実現した．

提案手法では，copy on write を抑制して命令の書き込みを行っているため，書き込みによるアップデート内容はファイルキャッシュに残る．しかしこのファイルキャッシュは，通常のファイル書き込み命令によって作られたものではないため，書き戻し機構によって実行ファイルに書き込まれない．そこで，更新対象プロセスのファイルキャッシュがディスクに書き戻されるようにするために，当該ファイルキャッシュが書き戻し対象となるように `buffer_head` 構造体を生成し，アップデート対象となったページと `buffer_head` 構造体を結びつけることで書き戻しを行うよう実装した．

表 1 評価環境

OS	Arch Linux (Linux Kernel 3.14.4)
CPU	Intel Core i5 760 (4 コア，2.8GHz)
メモリ	8GB

5. 評価

実装した提案手法の評価を行った．本章では copy on write の抑制の動作確認と，アップデート処理時の停止時間について既存手法と提案手法を比較する．評価環境を**表 1** に示す．

5.1 動作確認

copy on write の抑制が正しく行われているか動作確認を行った．拡張領域を持つテストプログラムを，更新対象プロセスとした．テストプログラムでは指定された数の子プロセスを生成し，親子プロセスが更新対象関数を繰り返し実行する．この更新対象関数は，何も処理を行わずに返り値として 0 を返すだけである．更新対象関数の返り値は printf 関数によって確認する．提案手法を用いて更新対象関数の返り値が 1 となるように親プロセスに対して動的アップデートを行った時，子プロセスもアップデートされるかを確認した．

実験の結果，親子プロセス共にアップデートされていることを確認した．これによって，実装を行った copy on write の抑制は正しく動作しており，動的アップデートに適用できていることがわかった．

5.2 停止時間の測定

既存手法と提案手法で，動的アップデート時の更新対象プロセスの停止時間を測定した．既存手法は livepatch を参考に実装した．既存手法による動的アップデートの手順は，プロセスを ptrace システムコールで停止させた後，mmap2 システムコールによって heap 領域を確保する．その後，heap 領域に更新済関数を書き込み，更新対象関数の先頭に jmp 命令を書き込む．この既存手法では，プロセスを一体ずつ動的アップデートするため，親子プロセスがアップデート対象である場合，各プロセスを順番に動的アップデートする必要がある．この場合，既存手法による更新対象プロセスの停止時間はプロセス数に比例して大きくなると考えられる．

停止時間の測定のために子プロセスの数を 1 から 1024 まで変化させ，各子プロセス数の環境で 100 回の動的アップデートを行った．その際に得られた停止時間の中央値 80 個の平均を算出した．**図 8** と**図 9** は算出した停止時間の平均であり，**図 10** は，算出した停止時間の平均を既存手法を 1 として正規化したものである．各グラフの横軸は動的アップデート対象の子プロセス数を表している．

Vol.2014-DBS-160 No.2
Vol.2014-OS-131 No.2
Vol.2014-EMB-35 No.2
2014/11/18

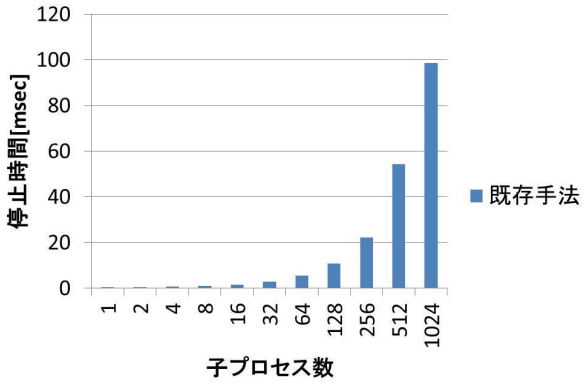

図 8　既存手法と提案手法を比較した評価結果

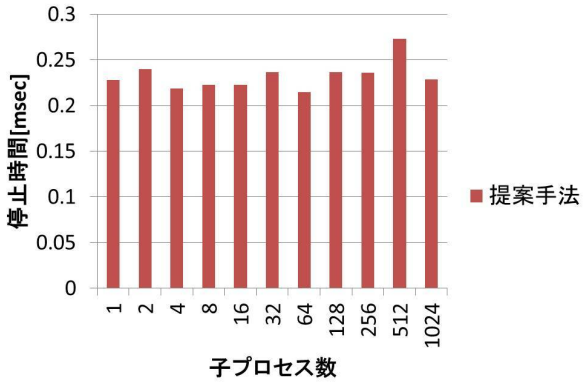

図 9　提案手法を適用した場合のプロセスの停止時間

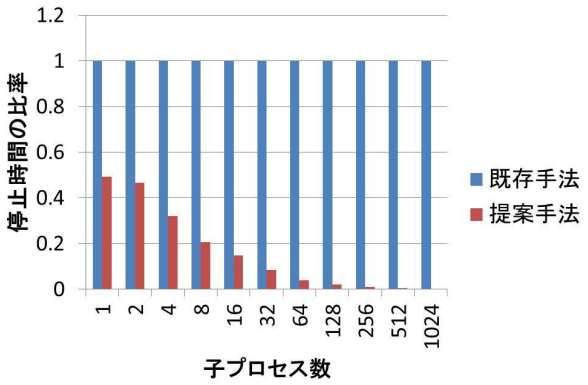

図 10　既存手法を 1 として正規化した結果

既存手法の評価結果を図 8 に，提案手法の評価結果を図 9 に示す．既存手法の停止時間は子プロセス数に比例して長くなるのに対して，提案手法の停止時間は子プロセス数に関係なくほぼ一定であることが分かる．また，提案手法の停止時間はどの子プロセス数においても，既存手法より短い．

図 10 より，子プロセスの数が増えるごとに提案手法の値が小さくなっていることがわかる．これは，既存手法が子プロセスの数に比例して停止時間が増加するのに対して，提案手法がプロセス数に関係なく一定の停止時間であるためである．

以上より提案手法による，同一バイナリの複数プロセスを対象とする動的アップデートでは，プロセス数に限らずほぼ一定の停止時間で動的アップデートが実現できることを確認した．また，既存の動的アップデート手法に比べ，少ない停止時間で動的アップデートが実現できることも確認した．

6. まとめ

本稿では，同一バイナリの複数プロセスにおける動的アップデートの停止時間削減手法を提案した．提案手法は，copy on write を動的アップデート時に抑制することで，プロセス間で共有されているページに書き込みを行うことを実現した．これによって単一プロセスのアップデート内容が同一バイナリの複数プロセスすべてに共有されるため，同時に動的アップデートを行うことが可能となる．提案手法による停止時間を測定し，停止時間がプロセス数によらずほぼ一定であり，既存の動的アップデート手法に比べ小さいことを確認した．今後は実アプリケーションにおける動作確認を行う予定である．

参考文献

[1] Lee, I.: Dymos: a dynamic modification system, PhD Thesis, University of Wisconsin–Madison (1983).
[2] Apache HTTP SERVER PROJECT: `http://httpd.apache.org/`.
[3] Tetu Takabayasi, H. U.: Binary Hacks ハッカー秘伝のテクニック 100 選，OREILLY (2006).
[4] Arnold, J. and Kaashoek, M. F.: Ksplice: Automatic rebootless kernel updates, *Proceedings of the 4th ACM European conference on Computer systems*, pp. 187–198 (2009).
[5] Oracle Linux: `http://www.oracle.com/jp/technologies/linux/overview/index.html`.
[6] NTT Corporation: `http://pannus.sourceforge.net/`.
[7] Yamato, K., Abe, T. and Corpration, M.: A runtime code modification method for application programs, *Proceedings of the Ottawa Linux Symposium* (2009).
[8] Chen, H., Yu, J., Chen, R., Zang, B. and Yew, P.-C.: Polus: A powerful live updating system, *Proceedings of the 29th international conference on Software Engineering*, pp. 271–281 (2007).