

走行中プロセスを 他プロセスから複製する機能の提案

鈴木 森羅¹ 乃村 能成¹ 谷口 秀夫¹

概要：走行中プロセスを他プロセスから複製する機能により、走行中プロセスの途中状態を保存したり、異なる入力に対する動作を観察したりできる。例えば、ソフトウェアのテストのような特定の状態のプロセスに様々な入力を与えたい場面において、有用である。また、保存したプロセスの途中状態から再びプロセスを走行させることにより、入力情報を操作してプロセスの状態を再現することもできる。本稿では、走行中プロセスを他プロセスから複製する機能を提案し、評価結果を報告する。

1. はじめに

走行中プロセスを他プロセスから複製できれば、複製によって生成されたプロセスを走行中プロセスの途中状態として保存したり、異なる入力を与えて動作を観察したりできる。これは、例えば、ソフトウェアのテストにおいて、有用である。ソフトウェアのテストを行う際、特定の状態のプロセスに様々な入力を与える。このような場面では、与える入力の数に応じて、プロセスの状態を繰り返し再現する必要がある。そこで、保存した途中状態を用いることで、プロセスの状態を容易に利用できる。また、フォールトトレランスや負荷分散においても有用である。走行中のプロセスと同じプロセスを1つ以上走行させることでプロセスに冗長性を持たせられる。走行中プロセスを他プロセスから複製できれば、容易にプロセスを冗長化できる。さらに、既に提案されているゲームの途中状態を複製・共有するシステム [1] における、ゲームの複製への利用も考えられる。

先行研究として、アプリケーションのスナップショットを保存し、スナップショットからアプリケーションの走行を再開させる技術はチェックポイント/リスタート (以降、C/R) と呼ばれ、研究されている。Linux 上では、DMTCP[2] や BLCR[3] といった実装例が存在する。また、BLCR を基に、MPI を利用した並列分散システムの C/R 手法も実現されている [4]。さらに、この C/R 手法を用いることで、プロセスの耐障害性を高めるライブマイグレーション手法も実現されており、Proactive[5] と呼ばれる。このように

C/R は並列分散システムの耐障害性を目的として使用されている。C/R では、プロセスの持つ様々な情報を保存し、その情報を用いてプロセスを再開させる。一方、我々は、走行中プロセスを複製し、生成されたプロセス自体を途中状態として保存する。そして、保存した途中状態を基にプロセスを再開させることで、上記のような場面での利用を目指す。そこで、本稿では、走行中プロセスを他プロセスから複製する機能 (以降、プロセス複製機能) を提案する。

UNIX オペレーティングシステムでは、プロセスを複製するシステムコールとして `fork()` システムコールを用意している。しかし、`fork()` システムコールは発行したプロセス自身を複製するシステムコールであるため、他のプロセスを複製できない。そこで、提案するプロセス複製機能では、複製の対象となるプロセス (以降、対象プロセス) のシステムコール発行によるカーネル空間への処理遷移時、対象プロセスに `fork()` システムコール相当の処理 (以降、`fork()` ルーチン) を実行させることで対象プロセスを複製する。

2. 走行中プロセスを他プロセスから複製する機能

2.1 要求条件

プロセス複製機能は、他プロセスからカーネルへ対象プロセスの複製を要求する。その後、対象プロセスのシステムコール発行によるカーネル空間への処理遷移時、対象プロセスに `fork()` ルーチンを実行させることで対象プロセスを複製する。そこで、プロセス複製機能実現のための要求条件を以下に示す。

(要求 1) 複製される対象プロセスを指定可能

¹ 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology,
Okayama University

対象プロセスの複製をカーネルへ要求する要求プロセスは、カーネルへ任意の走行中プロセスの複製を要求する。このため、要求プロセスは走行中のプロセスの中から対象プロセスとして任意のプロセスを指定できる必要がある。

(要求 2) 対象プロセスのソースコードの変更不要

対象プロセスのソースコードの変更を不要にすることで、商用のソフトのようなソースコードを変更できない走行中プロセスを複製の対象にできる。このため、対象プロセスのソースコードを変更することなく、複製する必要がある。

(要求 3) 対象プロセスと生成プロセスは独立に走行

対象プロセスと生成プロセスは互いの処理に影響を与えてはならない。そこで、対象プロセスと生成プロセスは独立に走行する必要がある。ここで、独立に走行とは、一方のプロセスの処理がもう一方のプロセスの処理へ影響を与えない状態である。影響を与える処理として、例えば、両プロセスが親子関係である際に一方のプロセスが動作を終了、また両プロセス間で入出力対象を共有している際に一方のプロセスが入出力を行うといった処理が挙げられる。

2.2 対処

各要求に対する対処を以下に示し、説明する。

(対処 1) PID を用いて対象プロセスを指定 (要求 1 への対処)

UNIX オペレーティングシステムにおいて、各プロセスは一意な識別子として PID を割り当てられている。そこで、複製を要求する際、要求プロセスは PID を用いて対象プロセスを指定する。

(対処 2) 対象プロセスのシステムコール発行を契機に複製 (要求 2 への対処)

(要求 2) のため、対象プロセスのソースコードを変更することなく、対象プロセスを複製する。そこで、対象プロセスのカーネル空間への処理遷移時、対象プロセスに `fork()` ルーチンを実行させる。ここで、プロセスの処理がカーネル空間に遷移する契機はいくつかある。その内の 1 つはシステムコールの発行である。プロセスはカーネルの機能を利用する際、必ずシステムコールを発行する。したがって、システムコールは多くのプロセスにおいて発行される。そこで、プロセス複製機能では、対象プロセスのシステムコール発行による、カーネル空間への処理遷移時、対象プロセスに `fork()` ルーチンを実行させる。

(対処 3) 対象プロセスと同一の親プロセスに設定 (要求 3 への対処)

`fork()` ルーチンによる複製において、対象プロセスと生成プロセスは独立に走行できない原因の 1 つは

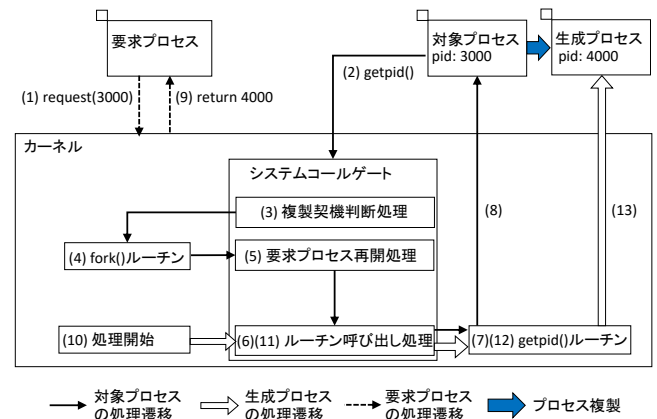


図 1 プロセス複製機能の処理流れ

両プロセスの親子関係である。

UNIX オペレーティングシステムでは、子プロセスは処理を終了する際、親プロセスへシグナルを送信する。このため、子プロセスの終了は親プロセスへ影響を与える。

Linux を含む多くの UNIX オペレーティングシステムでは、プロセスの複製を行う際、生成されたプロセスの親プロセスについて、複製元のプロセスにする、または複製元のプロセスの親と同一にするのどちらかを選択できる。このような機能は UNIX 系の多くのオペレーティングシステムに実装されている。例えば、Linux の場合、`clone()` システムコールを用いて複製を行う際、親プロセスについて、どちらかを選択して複製できる [6]。また、FreeBSD においても、同様に `clone()` システムコールによる複製の際、親プロセスについて、どちらかを選択できる [7]。

この機能を用いて、生成プロセスの親プロセスを対象プロセスの親と同一にする。これにより、対象プロセスと生成プロセスは同じ親プロセスを持つ。このため、対象プロセスと生成プロセスは親子関係を結ばない。

(対処 3) を行うのみでは、対象プロセスと生成プロセスは独立に走行できない。このため、(対処 3)に加えて、(要求 3) への対処を検討する必要がある。

2.3 処理流れ

プロセス複製機能の処理流れを図 1 に示し、以下で説明する。図 1 では、要求プロセスからカーネルへ対象プロセスの複製を要求する。そして、対象プロセスは例えば `getpid()` システムコールの発行により、カーネル空間へ処理遷移する。これを契機に対象プロセスは `fork()` ルーチンを実行し、自身を複製する。ここで、対象プロセス、生成プロセスの PID をそれぞれ 3000、4000 とする。

(1) 対象プロセスの複製を要求するため、要求プロセスは複製要求システムコールを発行する。複製要求シス

テムコールとは、カーネルへ対象プロセスの複製を要求するためのシステムコールである。このとき、複製要求システムコールの引数として対象プロセスのPID 3000 を渡し、複製の対象を指定する。その後、返り値として生成プロセスのPID を取得するため、複製完了まで要求プロセスは処理を停止させ、待機する。

- (2) 対象プロセスはシステムコール(ここでは `getpid()`) 発行により、カーネル空間へ処理遷移する。以降、対象プロセスの処理はカーネル空間にて行われる。
- (3) 対象プロセスは複製要求の際、要求プロセスの指定したPIDと自身のPIDを比較する。これにより、2つの値の一致を以て、対象プロセスは自身の複製を要求されていると判断する。そして、複製を要求されていれば、`fork()` ルーチンを呼び出す。
- (4) 対象プロセスは `fork()` ルーチンを実行し、自身を複製する。これにより、生成プロセスは生成される。このとき、対象プロセスと生成プロセスの親子関係を解消する。また、`fork()` ルーチンの返り値である生成プロセスのPID 4000 を保存する。
- (5) 対象プロセスの複製は完了したため、要求プロセスの処理を再開させる必要がある。そこで、対象プロセスは要求プロセスをスケジューリングする。
- (6) 対象プロセスは(2)で発行された `getpid()` システムコールの処理(以降、`getpid()` ルーチン)を呼び出す。
- (7) 対象プロセスは `getpid()` ルーチンを実行する。
- (8) 対象プロセスの処理は `getpid()` ルーチンの返り値と共にカーネル空間からユーザ空間へ帰還する。
- (9) 対象プロセスの複製完了後、要求プロセスはディスパッチされる。すると、要求プロセスは保存されている生成プロセスのPID 4000 を取得し、ユーザ空間へ帰還する。
- (10) 生成プロセスは処理を開始する。
- (11) 生成プロセスにおいても複製の契機となった `getpid()` システムコールは発行されたことになっている。このため、生成プロセスも対象プロセスと同様に `getpid()` ルーチンを呼び出す。
- (12) 生成プロセスは `getpid()` ルーチンを実行する。
- (13) 生成プロセスの処理は `getpid()` ルーチンの返り値と共にカーネル空間からユーザ空間へ帰還する。

2.4 実現における課題と対処

2.4.1 課題

2.3 節で示した設計を実現するうえで、課題となるのは、対象プロセスと生成プロセスによる複製契機となったシステムコールの処理(以降、契機ルーチン)の実行である。2.3 節の処理流れでは、対象プロセスと生成プロセスはそれぞれ契機ルーチンを実行する。しかし、以下の原因から両プ

ロセスは契機ルーチンをそのまま実行すると、不具合が起きる。

- (1) プロセス複製機能のために追加された処理により書き換えられたレジスタ

通常、発行されたシステムコールに応じた処理を実行する際、レジスタにその処理へ渡す引数や戻りアドレスといった情報を保持している。しかし、プロセス複製機能のために追加された処理の実行により、対象プロセスの複製を完了したとき、レジスタの値は書き換えられ、それらの情報を保持していない。このため、契機ルーチンを実行したとしても、正しく実行されない。

- (2) 生成プロセスの `ret_from_fork()` 実行後の処理遷移

`ret_from_fork()` は `fork()` ルーチンによって生成されたプロセスの最初に実行する関数である。そして、`fork()` ルーチンによって生成されたプロセスの処理は、通常、`ret_from_fork()` を実行後、システムコールゲートのルーチン呼び出し処理以後へ遷移する。このため、生成プロセスは契機ルーチンを呼び出せない。

2.4.2 対処

2.4.1 項で示した課題への対処を以下に示し、説明する。

- (対処 1) カーネルスタックに保存された値を基にレジスタを復元

プロセスの処理はシステムコールゲートに遷移すると、まず、レジスタの値をカーネルスタックに保存する。そこで、契機ルーチンの実行前に、書き換えられたレジスタの値をカーネルスタックの値を基に復元する。この方法により、対象プロセスのレジスタの値は復元できる。しかし、生成プロセスのレジスタの値は復元できない。

この原因は `fork()` ルーチンの返り値の返却方法にある。`fork()` システムコールは親プロセスに子プロセスのPIDを、子プロセスに0を返す。このため、`fork()` ルーチンは親プロセスと子プロセスの両方に返り値を返す必要がある。しかし、1つのルーチンは2つの返り値を返せない。そこで、`fork()` ルーチンは子プロセスのカーネルスタックを書き換えることで子プロセスへ返り値を返す。具体的には、システムコール番号を保存したカーネルスタックのエントリに子プロセスへの返り値を保存する[8]。したがって、対象プロセスの `fork()` ルーチン実行により、生成プロセスのカーネルスタックに保存されたシステムコール番号は書き換えられる。このため、生成プロセスのレジスタの値はカーネルスタックの値のみでは復元できない。そこで、対象プロセスの `fork()` ルーチン実行前に契機ルーチンのシステムコール番号を保存しておく。そして、対象プロセスの複製完了後、保存したシステムコール番号とカーネルスタックの値を基にレジスタの

値を復元する。

(対処 2) `ret_from_fork()` の処理遷移先の変更

`ret_from_fork()` の処理遷移先を変更し、生成プロセスの処理は `ret_from_fork()` の実行後、契機ルーチンの呼び出し前に遷移するようにする。ただし、`ret_from_fork()` は通常の `fork()` システムコールにより生成されたプロセスも実行する。このため、プロセス複製機能による生成プロセスの実行時のみ、実行後、システムコールゲートの契機ルーチン呼び出し前に処理遷移するように `ret_from_fork()` を変更する。これらの対処を行うことで、プロセス複製機能を実装した。

3. 評価

3.1 観点

本評価では、プロセス複製機能について以下の観点から評価を行う。

(観点 1) システムコールに加わるオーバーヘッド

カーネルを変更し、プロセス複製機能を実装することでプロセスの発行する全てのシステムコールにおいて、プロセスは自身の複製を要求されているか否か確認する。このため、機能実装前のカーネルに比べて実装後のカーネルでは、全てのシステムコールに多少のオーバーヘッドが加わる。そこで、このオーバーヘッドは十分に小さいものであるか否かを評価する。

(観点 2) 対象プロセスに与えるコピーオンライトの影響

プロセス複製機能により対象プロセスを複製すると、複製後の対象プロセスの処理は複製前に比べて、遅くなる。なぜなら、対象プロセスは複製処理以後に未更新のページへ書き込みを行う際、コピーオンライトにより、当該ページのコピー処理を実行するためである。これは生成プロセスの走行状態に関わらず、発生する。そこで、複製前後のメモリ書き込みの処理時間を比較し、対象プロセスの処理時間に与えるコピーオンライトの影響を評価する。

(観点 3) プロセス複製機能の実装によるコード変更量

プロセス複製機能の導入工数が多ければ、複製したいプロセスごとにソースコードを変更し、`fork()` システムコールを挿入する方が手軽になってしまう。そこで、プロセス複製機能のためのコード変更量をまとめ、これは大きいのか小さいのか評価する。

3.2 環境

評価環境を表 1 に示す。

3.3 システムコールに加わるオーバーヘッド

3.3.1 測定方法

プロセス複製機能の実装によりシステムコールに加わるオーバーヘッドを評価する。そこで、プロセス複製機能実装

表 1 評価環境

項目名	環境
OS	Debian 7.10
カーネル	Linux 3.0.8 64bit
CPU	Intel(R) Core(TM) i7/2.93GHz
メモリ	2GB
ページサイズ	4096B

前後のカーネルにおけるシステムコールの処理時間を測定する。ここでは、`getpid()` システムコールを測定対象とする。なお、システムコールの発行から返り値の取得までを処理時間とする。

3.3.2 結果と考察

3.3.1 項の測定の結果を表 2 に示す。表 2 によると、プロセス複製機能の実装により全てのシステムコールには 0.04 マイクロ秒のオーバーヘッドが加わる。これは変更前のカーネルにおけるシステムコールの処理時間に対して、`getpid()` システムコールの場合、約 6 % である。`getpid()` システムコールは最も計算量の小さいシステムコールであるため、システムコールにかかるオーバーヘッドは最大約 6 % である。

表 2 プロセス複製機能実装前後のシステムコールの処理時間の測定結果 (マイクロ秒)

カーネルの状態	処理時間
実装前	0.74
実装後	0.78
オーバーヘッド	0.04

3.4 対象プロセスに与えるコピーオンライトの影響

3.4.1 測定方法

本測定では、メモリ書き込みを行うプロセスを対象プロセスとして、プロセス複製機能を用いて複製する。そして、複製前後のメモリ書き込みの処理時間を測定する。ここで、対象プロセスの処理内容を以下に示す。

- (1) 書き込みを行うメモリ領域を確保する。今回、確保するメモリ領域の大きさは 4MB とした。
- (2) 確保したメモリ領域の全域を 0 で埋め、初期化する。
- (3) 確保したメモリ領域に以下のどちらかの書き込み方法で書き込みを行う。

(シーケンシャル書き込み) 4MB の領域の先頭から末尾まで 1 回通して 4B ずつ書き込む。つまり、 1024×1024 回、4B のメモリ書き込みを行う。

(ランダム書き込み) 4MB の範囲に 4B ずつランダムに 1024×1024 回書き込む。

- (4) `getpid()` システムコールを発行する。これを契機にプロセス複製機能を用いて複製を行う。
- (5) メモリ領域に (3) の処理と同様の書き込み方法で書き込みを行う。

この処理内容の内、(3)と(5)の処理の直前と直後でタイムスタンプを取得し、複製前後のメモリ書き込み全体の処理時間を測定する。また、(5)の処理中に行う 1024×1024 回のメモリ書き込みの内、各ページへ最初に書き込みをした際のメモリ書き込み処理の直前と直後でタイムスタンプを取得し、コピー処理の加わったメモリ書き込み時間を測定する。このとき、何回目の書き込みであるかも記録する。なお、(3)と(5)の処理内容を揃えるため、このタイムスタンプの取得は(3)においても行う。

3.4.2 結果と考察

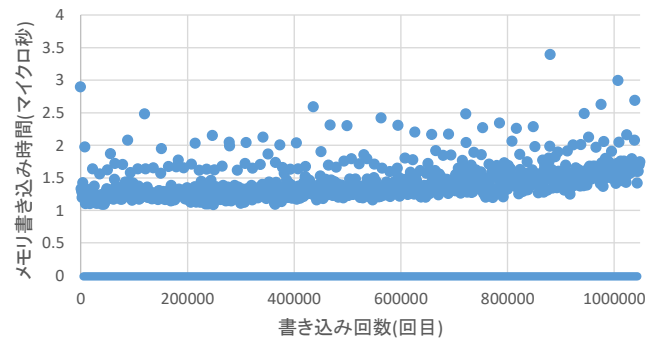
3.4.1 項の測定の結果を表 3 に示す。表 3 では、対象プロセスが複製前後に行う 4MB の範囲へ 4B ずつ 1024×1024 回のメモリ書き込みの処理全体の処理時間を示している。表 3 によると、シーケンシャル書き込みとランダム書き込みともに複製後の処理時間は増加した。このため、メモリ書き込みを頻繁に行うプロセスであれば、複製後のメモリ書き込みにおいて、処理時間は同程度増加すると考えられる。

表 3 複製前後の対象プロセスのメモリ書き込み処理全体の処理時間 (ミリ秒)

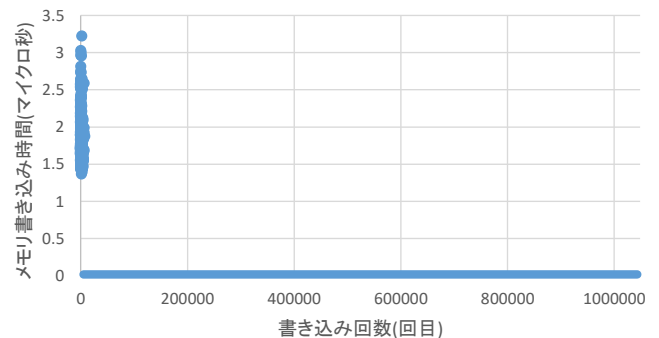
	シーケンシャル書き込み	ランダム書き込み
複製前	17.3	18.9
複製後	19.2	21.1
増加した時間	1.8	2.2
増加した割合	10.6 %	11.8 %

また、4MB の範囲に 4B ずつ 1024×1024 回のメモリ書き込みを行う中で、何回目の書き込みでコピー処理が発生しているのかを調査する。そこで、縦軸にメモリ書き込み時間、横軸に書き込み回数を取り、シーケンシャルとランダムそれぞれの書き込み方法における 1024×1024 回のメモリ書き込み全てをプロットする。このとき、コピー処理の発生しないメモリ書き込み時間は極めて短いため、メモリ書き込み時間は 0 としてプロットする。プロットの結果を図 2 に示す。ここで、図 2 はデータ数が多く、読み取りにくいので、データを 1 回目から 5000 回目までのメモリ書き込みに絞り、プロットする。プロットの結果を図 3 に示す。図 3 を見ると、シーケンシャル書き込みでは、一定の間隔でコピー処理が発生し、ランダム書き込みでは、集中して発生している。これにより、全体の処理時間にどのような影響を与えるのかを調査する。

そこで、図 2 と図 3 のデータを基に、書き込み回数に応じて、累積メモリ書き込み時間はどのような推移で伸びているのかを図 4 と図 5 に示す。図 4 によると、シーケンシャル書き込みとランダム書き込みともに累積メモリ書き込み時間は線形に伸びている。ただし、ランダム書き込みの書き込み開始直後に注目すると急激に累積メモリ書き込み時間が伸びている。そこで、図 5 から書き込み開始直後

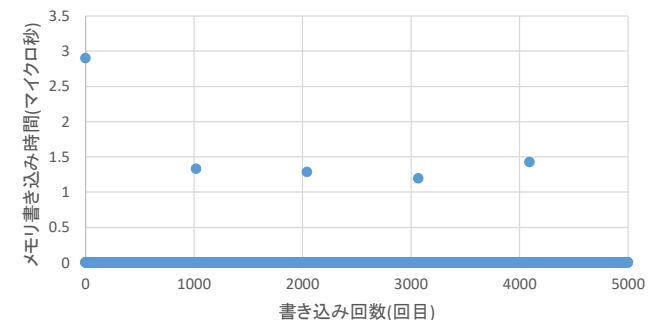


(A) シーケンシャル書き込み

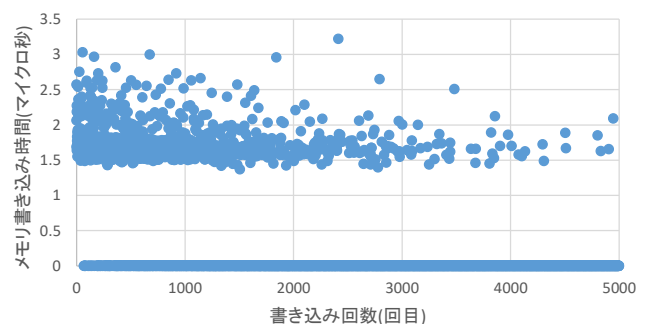


(B) ランダム書き込み

図 2 4B ずつのメモリ書き込み時間 (1024×1024 回目まで)



(A) シーケンシャル書き込み



(B) ランダム書き込み

図 3 4B ずつのメモリ書き込み時間 (5000 回目まで)

の累積メモリ書き込み時間の推移を見る。すると、ランダム書き込みでは、コピー処理は書き込み開始直後に頻繁に発生するため、急激に累積メモリ書き込み時間が伸びている。そして、4000 回ほど書き込みを行うと、コピー処理の

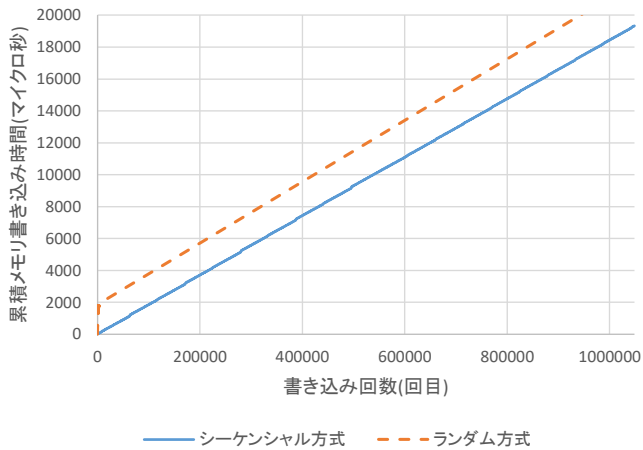


図 4 累積メモリ書き込み時間 (1024 × 1024 回目まで)

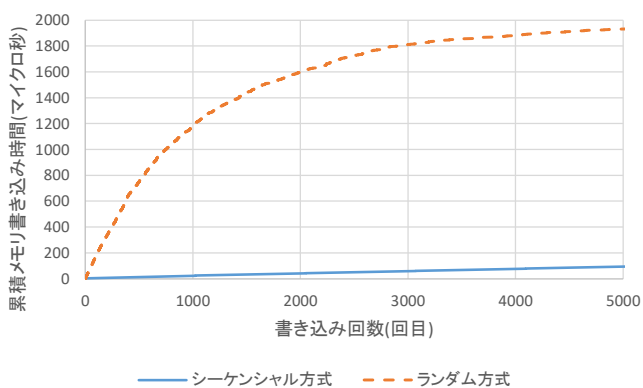


図 5 累積メモリ書き込み時間 (5000 回目まで)

発生が少なくなり、累積メモリ書き込み時間の伸びが緩やかになっている。一方、シーケンシャル書き込みでは、一定の間隔でコピー処理が発生する。このため、累積メモリ書き込み時間への影響は急激に出ることはなく、1 回目から 5000 回目まで通して、線形に累積メモリ書き込み時間が伸びている。5000 回目の時点での累積メモリ書き込み時間を見ると、ランダム書き込みはシーケンシャル書き込みの約 20 倍である。このため、メモリ書き込みを頻繁に行うプロセスを複製すると、ランダム書き込みほどではないが、コピー処理は複製直後に集中して発生し、複製直後の処理時間は伸びると考えられる。

最後に、対象プロセスに与えるコピーオンライトの影響をまとめる。メモリ書き込みを頻繁に行うプロセスであれば、複製後の処理時間は、複製前に比べて、約 10 % 程度増加する。また、同様のプロセスを複製すると、ランダム書き込みほどではないが、コピー処理は複製直後に集中して発生し、複製直後の処理時間も伸びると考えられる。ただし、多くのプログラムでは、メモリ書き込み以外の処理も多く行うため、処理時間の増加する割合はより小さくなり、コピー処理の発生もより分散する。したがって、多くのプロセスを複製する場合、対象プロセスに与えるコピーオンライトの影響は小さいと言える。

3.5 プロセス複製機能の実装によるコード変更量

3.5.1 測定方法

プロセス複製機能を実装するための変更量を明らかにする。なお、本評価では、コード量の評価基準として、論理 LOC (Lines Of Code) を使用する。論理 LOC は、ソースコードの総行数から、記号のみの行、空白行、およびコメントのみの行を省いた総数である。論理 LOC の算出には、CLOC[9] を使用する。

表 4 プロセス複製機能における変更量

通番	ファイル名	各変更箇所の追加行数				
		A	B	C	D	合計
1	arch/x86/include/asm/syscalls.h	2	-	-	-	2
2	arch/x86/include/asm/unistd_64.h	4	-	-	-	4
3	arch/x86/kernel/Makefile	1	-	-	-	1
4	arch/x86/kernel/entry_64.S	-	23	-	7	30
5	arch/x86/kernel/process.c	-	-	2	-	2
6	arch/x86/kernel/sys_request_fork.c	16	-	-	-	16
合計	6 ファイル	23	23	2	7	55

3.5.2 結果と考察

測定の結果、変更したファイルと各ファイルに追加した行数を表 4 に示す。ただし、通番 6 のファイルはプロセス複製機能を実装するために追加したファイルである。

また、プロセス複製機能の実装により変更された箇所を以下に示す。なお、各項目の末尾に表 4 の通番を用いて変更に関わるファイルを示す。

(変更箇所 A) 複製要求システムコール (通番 1, 2, 3, 6)

要求プロセスからカーネルへ対象プロセスの複製を要求するため、複製要求システムコールをカーネルへ追加した。

(変更箇所 B) システムコールゲート (通番 4)

プロセス複製機能において、システムコールゲートはプロセスの発行するシステムコールを監視し、対象プロセスのシステムコール処理中に `fork()` ルーチンを実行する。その後、複製完了を待機している要求プロセスの処理を再開させる。最後に、契機ルーチンを実行するため、`fork()` ルーチンの実行により書き換えられたレジスタを復元する。システムコールゲートを変更し、以上の処理を追加した。

(変更箇所 C) `fork()` ルーチン (通番 5)

`fork()` ルーチン内の `do_fork()` の引数を変更し、生成プロセスの親プロセスを対象プロセスの親と同一にするようにした。

(変更箇所 D) `ret_from_fork()` (通番 4)

プロセス複製機能の生成プロセスによる実行後の処理遷移先をシステムコールゲート内のシステムコールのルーチン呼び出し処理以前に変更した。

表 4 によると、変更量は全 6 ファイルに渡って、55 行のコードである。これは Linux カーネル全体のファイル数とコード量に比べて極めて小さい。このため、変更箇所を局所化し、変更量も少なくできていると考える。したがって、プロセス複製機能の導入は容易であると言える。

4. おわりに

走行中プロセスを他プロセスから複製する機能を提案した。この機能では、他プロセスからカーネルへ走行中プロセスの複製を要求する。その後、複製を要求されたプロセスのシステムコール発行によるカーネル空間への処理遷移時、複製を要求されたプロセスに `fork()` ルーチンを実行させることでプロセスを複製する。

提案機能の実現に対する要求として、複製される対象プロセスを指定可能、対象プロセスのソースコードの変更不要、および対象プロセスと生成プロセスは独立に走行という 3 つを示した。これらの要求への対処として、PID を用いて対象プロセスを指定、対象プロセスのシステムコール発行を契機に複製、および対象プロセスと同一の親プロセスに設定を示した。また、実装上の課題として、対象プロセスと生成プロセスそれぞれにおける契機ルーチンの実行を挙げた。この対処としてカーネルスタックに保存された値を基にレジスタを復元と `ret_from_fork()` の処理遷移先の変更を示した。

評価により、プロセス複製機能として、システムコールに加わるオーバーヘッドは最大約 6 % であること示した。また、複製後、対象プロセスに与えるコピーオンライトの影響は小さいことを示した。

参考文献

- [1] 市川優平, 乃村能成: ゲームの途中状態を複製・共有するシステムの提案, 情報処理学会第 78 回全国大会講演論文集, Vol. 3, pp. 23–24 (2016).
- [2] Ansel, J., Arya, K. and Cooperman, G.: DMTCP: Transparent checkpointing for cluster computations and the desktop, *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE, pp. 1–12 (2009).
- [3] Hargrove, P. H. and Duell, J. C.: Berkeley lab checkpoint/restart (blcr) for linux clusters, *Journal of Physics: Conference Series*, Vol. 46, No. 1, IOP Publishing, p. 494 (2006).
- [4] Sankaran, S., Squyres, J. M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P. and Roman, E.: The LAM/MPI checkpoint/restart framework: System-initiated checkpointing, *International Journal of High Performance Computing Applications*, Vol. 19, No. 4, pp. 479–493 (2005).

- [5] Wang, C., Mueller, F., Engelmann, C. and Scott, S. L.: Proactive process-level live migration in HPC environments, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, p. 43 (2008).
- [6] Man page of CLONE, JM Project (online), available from https://linuxjm.osdn.jp/html/LDP_man-pages/man2/clone.2.html (accessed 2017-4-11).
- [7] FreeBSD Man Pages, FreeBSD (online), available from <https://www.freebsd.org/> (accessed 2017-4-11).
- [8] Daniel P. Bovet, M. C.: *Understanding the Linux Kernel, 3rd Edition*, O'Reilly Media (2005).
- [9] CLOC – Count Lines of Code, sourceforge.net (online), available from <http://cloc.sourceforge.net/> (accessed 2017-1-27).