Science Journal of Kanagawa University **18**：37-43 (2007)

■原　著■

Fork システムコールを用いた並列ガーベジコレクションの Lisp 処理系への実装と評価

佐藤憲一郎 [1,3]　松井祥悟 [2,4]

Implementation of Parallel Garbage Collection Using Fork System Call in Lisp System and Its Evaluation

Kenichiro Sato[1,3] and Shogo Matsui[2,4]

[1] Graduate School of Information Science, Kanagawa University, 2946 Tsuchiya, Hiratsuka-shi, Kanagawa 259-1293, Japan
[2] Department of Information Science, Kanagawa University, 2946 Tsuchiya, Hiratsuka-shi, Kanagawa 259-1293, Japan
[3] Presently with Hitachi Information Systems. Ltd.
[4] To whom correspondence should be addressed. E-mail: sho@info.kanagawa-u.ac.jp

Abstract: We implemented the parallel garbage collector using fork system call for process generation supported in the multi-processes OS, suchas UNIX. The GC does not need a write barrier because the gc process performs marking the replica space generated by fork system call. Although Inter-Process Communication is needed in order tonotify garbagecells, the ordinary stop and collect mark-sweep GC can be easily changed to this method. We implemented the GC in the Lisp system and compared with the original mark-sweep GC.

Keywords: garbage collection, parallel gerbage collection, garbage collection using fork system call

序論

動的なデータ構造を取り扱うシステムでは、ヒープ領域に配置したメモリ資源をプログラムが必要に応じて領域確保し使用する、という形態のデータ処理を行うのが一般的である。このようなシステムでは、使用済みのメモリを自動的に回収するガーベジコレクション（GC）の機構が不可欠である。

通常 GC は、割り当て可能なメモリが枯渇した時点で、メモリを消費して計算を行うプログラム（muta-tor）を停止し、メモリ回収プログラム（collector）を実行して使用済みのメモリを回収する停止-回収（stop-and-collect）方式で行う。

この停止-回収方式のアルゴリズムには、マークスイープ法やコピー法があるが、使用中のメモリを判別するためにヒープ領域をトレースする処理を行う点が共通している。この場合、mutator の停止時間はヒープ領域の大きさやトレースすべきメモリ量に比例して長くなる。この停止時間を短縮するために、世代 GC や並列 GC が開発されている。

並列 GC は、mutator と collector を並列に動作させる。完全な並列動作が実現できれば、collector による mutator の中断を完全に取り除くことができる。mutator が無停止になるので実時間処理への応用が可能となる。また、全体の処理時間が GC による時間分だけ短縮されるので高速化にもつながる。並列 GC の効率を改善する研究も行われている [5,7,10]。

並列 GC は、mutator によるセルの書き換えを collector へ通知するバリアが必要となる。そのため、停止-回収型 GC を並列 GC へ変更するには、mutator がセルを書き換える箇所へバリアを設定しなければならない。しかし、mutator のメモリ空間をコピーしたレプリカ空間に対して、collector が回収処理を行うことでバリアが不要となる。Rodriguez-Rivera らは、fork システムコールを使用してレプリカ空間を生成する並列 GC について述べている [6]。彼らの GC は、malloc 関数や free 関数などのメモリ管理処理を独自のライブラリとして実装

している。一方我々は、この fork システムコールを用いたプロセス生成型並列 GC を Lisp 処理系へ直接組み込んだ[3]。そして、停止-回収型 GC との比較を行い、効率や問題点について評価した。

本論

並列 GC のアルゴリズム

並列 GC の基本的なアルゴリズムと、並列化において問題となる点をあげる。

並列 GC の動作

並列 GC はマークスイープ法やコピー法のような停止-回収型の GC を並列化したものである。並列 GC は、collector に独立した専用のプロセッサを割り当て、mutator との完全な並列処理により GC を行う。一台のプロセッサ上での疑似並列処理による漸次（Incremental）GC も並列 GC の一種である。

マークスイープ法を基にした並列 GC では、一つの GC サイクルは、以下の三つのフェーズで構成される。

(1) ルート挿入（root-insertion）
(2) 印付け（marking）
(3) 回収（sweep）

ルート挿入フェーズでは、collector は mutator のレジスタやグローバル変数、スタック領域から mutator が使用中のセルを指すポインタをすべて収集する。これらの領域をルートセットと呼ぶ。印付けフェーズでは、収集したポインタから到達可能なすべてのセルに印を付ける。回収フェーズでは、全セルを走査し、印のないセルをフリーリストへ戻す。

バリア

停止-回収型 GC を単純に並列化した GC を考えると、collector の印付けやコピー中に mutator がセルのポインタを書き換えた時、使用中のセルに印が付かず、誤って回収される場合がある。それを防ぐために、mutator によるセルの書き込みに対してバリアを設け、collector へ通知する必要がある[2]（図 1）。

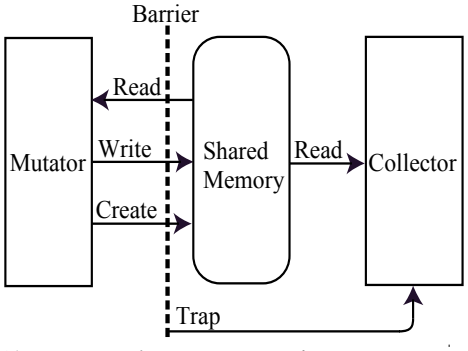

図 1. 並列ガーベジコレクタの構成とバリア.

基本的なアルゴリズム

一般的な並列 GC は、*incremental update*（IU）型と *snapshot-at-beginning*（SB）型の 2 種類に分類できる[9]。

IU 型 IU 型は停止-回収型 GC を単純に並列化したものである。collector は mutator のルートセットから到達できるセルに印付けを行い、印のないセルを回収してフリーリストへ戻す。停止-回収型との大きな違いは、collector による印付け中に mutator がセルの書き換えを行った場合の補償処理と、印付け終了後に必要なルートセット走査である。

SB 型 SB 型は GC 開始時にセルの状態をスナップショット写真を撮るように記録しておき、その記録上で使用済みとなったゴミセルだけを回収する。基本的には、スナップショット時にルートセットを含むすべてのセルの状態を記録し、それに基づいて印付けを行えばよい。実際にはルートセットだけを記録しておき、mutator がセルの書き換えを行う場合に、collector へ通知し、追加の印付けを行う。

スナップショット型並列 GC の動作

SB 型並列 GC は、はじめに mutator を停止してルートセットのコピーを行う（ルート挿入フェーズ）。その後、ルートセットに基づいて印付けを行う（印付けフェーズ）。この時に mutator によるセルの書き換えが発生すると、上書きされる古いポインタを collector へ通知する。ルートセットから到達できるすべてのセルへの印付けが終わると、mutator から通知されたセルへの追加の印付けを行う。最後に、印のない使用済みセルをフリーリストへ戻す（回収フェーズ）。mutator が停止するのは、ルート挿入フェーズでルートセットをコピーする時だけである。

純スナップショット型並列 GC

SB 型並列 GC は、通常スナップショット時にルートセットだけがコピーされるため、mutator によるセルの書き換えに対してバリアを設定する必要がある。しかし、セル空間全体を別の空間へコピーして、collector がこのコピーされた別空間のセルに対して印付けを行えば、バリアは必要なくなる。これを純スナップショット型 GC と呼ぶ[8]。

純スナップショット型 GC は、SB 型のスナップショット動作においてルートセットだけではなくセル空間全体を別の空間へコピーする（図 2）。collector は別空間のコピーされたセルに対して印付けを行う。この印にしたがって mutator のセル空間のゴミを回収し、フリーリストへ戻す。

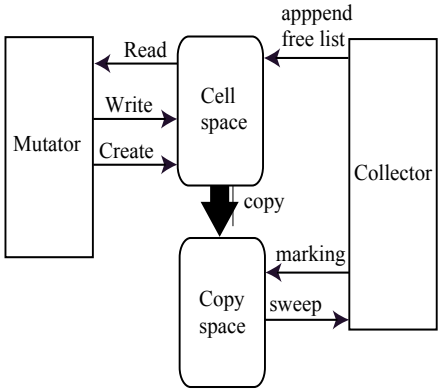

図 2. 純スナップショット GC の構成.

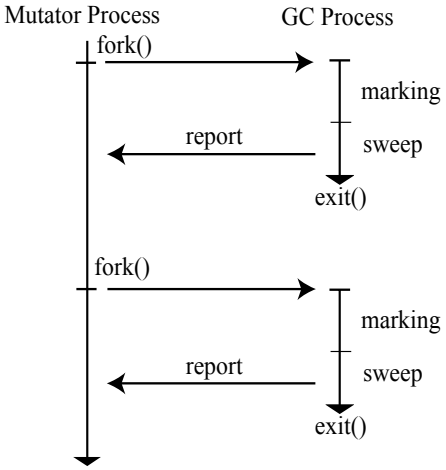

図 3. Fork GC の動作.

fork システムコールを用いた並列 GC

fork システムコールを用いた純スナップショット型 GC（Fork GC）[4]を図 3 に示す。fork()によりセル全体が子プロセス側へコピーされる。fork 実行後、親プロセスは mutator として通常の計算処理を続行する。子プロセスは collector として停止-回収型マークスイープと同じ GC 処理を行う。GC プロセスはスイープフェーズで再利用可能なゴミセル情報を親プロセスへ通知し、exit()で終了する。

コピーオンライト（copy-on-write, COW）型の fork システムコールの場合、プロセスを生成するときには、メモリ空間をコピーしたように見せかけておく。ページへの書き込みが発生すると、はじめて実際にコピー処理を行いメモリ空間を分離する。このように fork システムコールを用いることで、容易にセル全体のスナップショットを撮ることができる。

Rodriguez-Rivera らの並列 GC

Rodriguez-Rivera らの並列 GC のアルゴリズム[6]を以下に示す。

(1) 回収を起動したスレッド以外のスレッドを停止する。
(2) プロセスのメモリを複製する。
(3) すべてのスレッドを再開する。
(4) 子プロセス内の一つのスレッドを用いて印付けを行う。
(5) マークフェーズが終了するとマークビットを親プロセスへ渡して子プロセスを終了する。
(6) マークのないブロックをその後の割り当てに使用する（遅延スイープ）

彼らは、fork システムコールを用いた並列 GC を、共有ライブラリとして実装している。彼らの並列 GC は、malloc 関数や free 関数といったメモリ管理処理を独自のライブラリとして実装している。free 関数などの明示的にメモリを解放する関数ではメモリの解放を行わず、メモリの回収は GC だけで行う。

アプリケーション実行時に、ランタイムリンカがシステムライブラリ（libc）の代わりに独自に実装したメモリ管理ライブラリを動的にリンクするように指定している。

彼らの研究の目的の一つは、ソースコードが利用できないようなプログラムに対しても彼らの GC を使用できるようにすることである。そのために動的ライブラリとして実装することで、アプリケーションと独立した Nonintrusive（非侵害）性のある GC を実現している。

実験では、シングル CPU マシンおよびマルチ CPU マシン上で停止-回収型 GC や Boehm-Demers-Weiser GC（BDWGC）との比較を行っている。マルチ CPU マシン上においては、GC による総停止時間は停止-回収型や BDWGC より短くなるなど、良い結果が得られている。

fork システムコールを用いた並列 GC の実装

fork システムコールを用いた並列 GC（Fork GC）の実装について述べる。

mutator プロセスと GC プロセス間の通信

fork システムコールによって mutator プロセスと GC プロセスのメモリ空間は分離されるため、GC プロセスはスイープ時に mutator プロセスへゴミセル情報を通知する必要がある。主なプロセス間通信方法（IPC）にはパイプ、FIFO、メッセージキュー、共有メモリがある。その中でパイプや FIFO を用いるメッセージパッシング方法と共有メモリ方法について述べる。

(1) メッセージパッシング型
プロセス間でパイプなどの通信路を作成し、GC プロセスから mutator プロセスへゴミセル情報を送信する。

(2) 共有メモリ型
mutator プロセスと GC プロセスで情報を共有するためのメモリ領域を用意する。GC プロセスはスイープ時にゴミセル情報を共有メモリ上に書き込む。mutator プロセスはセルが必要となったときに共有メモリから取得する。つまり、共有メモリをフリーリストとして利用する。

プロセス間通信の違いによる得失

プロセス間通信の違いにより、次のような得失が生じる。
(1) mutator の停止
メッセージパッシング型の場合、GC プロセスからゴミセル情報の送信があると、mutator プロセスは処理を中断してフリーリストへゴミセルを追加しなければならない。この停止時間は GC プロセスが回収したセルの数（最悪の場合セル全体）に依存するので予測不可能である。共有メモリ型の場合には、このような mutator プロセスの停止はない。
(2) フリーリストの構成
メッセージパッシング型では、停止-回収型と同じ形式のフリーリストを用いることができる。共有メモリ型では、共有メモリ上にフリーリストを作成するように変更する必要がある。
(3) リソースの制限
共有メモリ型では、一つのプロセスが使用できる共有メモリのセグメント数と 1 セグメントあたりの最大メモリサイズに制限がある。例えば、SPARC 版 Solaris2.6 のデフォルトでは、1 プロセスに対して最大 6 セグメント、1 セグメントのメモリサイズは最大約 1 MB となっている。そのため、ヒープサイズが大きい場合には共有メモリ上にフリーリスト全体を作成することができなくなる。

評価

Lisp1.5 処理系で停止-回収型マークスイープ GC の euzak lisp に、パイプを用いたメッセージパッシング型（パイプ型）と、フリーリストをリングバッファ形式にした共有メモリ型の 2 種類の Fork GC を実装した。euzak lisp は C 言語で実装されており、UNIX 系 OS 上で動作する。評価を行った環境を表 1 および表 2 に示す。表 1 のマルチ CPU マシンの場合には、mutator プロセスと GC プロセスがそれぞれ 1 プロセッサを使用し、合計 2 プロセッサを用いている。評価に用いたベンチマークプログラムを以下に示す。boyer 論理式に対して論理証明を行うプログラムである。文献 1)のプログラムを使用した。

表 1. 評価環境（multiCPUs）.

CPU	HyperSPARC 100M Hz
CPU の個数	4
メインメモリ	512MB
OS	Solaris2.6
コンパイラ	GCC2.95.3

表 2. 評価環境（singleCPU）.

CPU	TurboSPARC 170MHz
メインメモリ	64MB
OS	Solaris2.6
コンパイラ	GCC2.95.3

表 3. ソースコードの変更量.

ForkGC の種類	ソースコードの総変更行数
パイプ型	170
共有メモリ型	163

(1) ack
原始帰納的でない帰納的関数で、2 つの引数が大きくなると計算量が膨大になる。
(2) hanoi
ハノイの塔を解くプログラムである。

ソースコードの変更量

表 3 に停止-回収型マークスイープ GC から並列型の ForkGC へ変更したときのソースコードの変更量を示す。

ソースコード全体は約 7700 行なので、変更は全体の約 2%である。主な変更箇所は、fork による GC プロセスの生成部分（約 100 行）である。変更量はパイプ型と共有メモリ型でほとんど変わらない。パイプ型の場合には mutator プロセスと GC プロセスの通信処理部分を追加しているため、共有メモリ型よりも若干多くなっている。

処理時間全体の評価

処理時間全体を計測し、比較を行った。処理時間は、euzaklisp を起動してフリーリストの作成などの初期化処理を行い、ベンチマークプログラムを実行し、処理系を終了するまでの時間である†。処理時間の計測には、シェル（tcsh）の組み込み関数の time を用いた。

停止-回収型のマークスイープ GC と並列型の Fork GC の処理時間を比較した結果を図 4 および図 5 に示す。図 4 はマルチ CPU マシン（表 1）、図 5

† 初期化処理などにかかる時間は数十ミリ秒で、処理時間全体の 0.1%以下なので無視している。

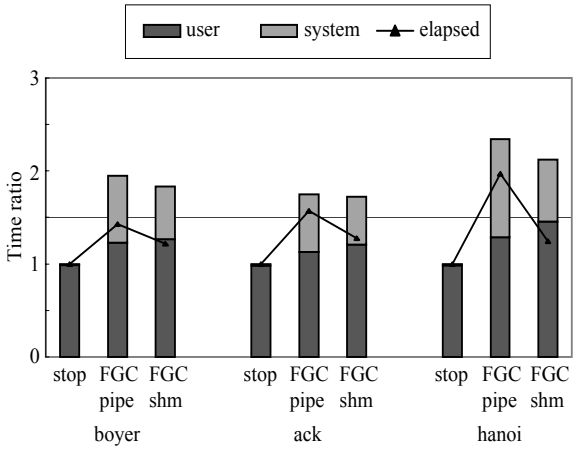

図 4. 停止-回収型と Fork GC の比較(multi CPUs).

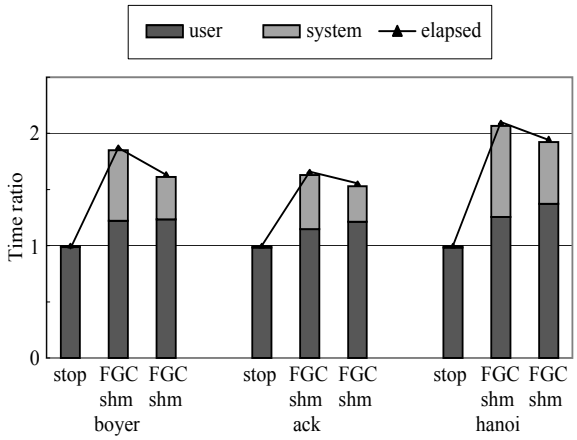

図 5. 停止-回収型と Fork GC の比較(single CPU).

はシングル CPU マシン（表 2）で計測を行った結果である。横軸は各 GC、縦軸は処理時間比である。処理時間が長いほど処理時間比は長くなる。

折れ線は処理経過時間（elapsed）で、棒グラフはユーザ時間（user）とシステム時間（system）を示している。停止-回収型マークスイープ GC（stop）の処理時間を 1 としている。マルチ CPU マシンの場合には、2 プロセッサによるユーザ時間とシステム時間の合計が処理経過時間を上回っているのは、複数のプロセッサによる並列処理のためである。

fork システムコールによるプロセス生成コストや、セルの書き換えによる COW におけるコピー処理のオーバーヘッドにより、停止-回収型よりも並列型 ForkGC の方が処理時間が長くなっている。システム時間が大幅に増えているのは、fork システムコールによるプロセス生成といったカーネルレベルでの処理量が増えたためと考えられる。

ForkGC のユーザ時間が増えているのは、プロセス間通信によるコストのためと考えられる。

また、パイプ型は mutator プロセスによるフリーリストへのゴミセルの追加やパイプによるプロセス間の非同期通信に時間がかかるため、処理時間全体では共有メモリ型の方が短くなっている。

シングル CPU マシン上でも疑似並列処理による漸次型 GC として動作している。しかし、mutator プロセスと GC プロセスは実際には並列に処理されないため、fork システムコールによるオーバーヘッドやセルの書き換え時の COW におけるコピー処理、パイプ型におけるプロセス間通信の時間分だけ処理時間全体が増えている。そのため、マルチ CPU マシン上よりもシングル CPU マシン上の方が処理経過時間の増加分が大きくなっている。

アッカーマン関数の計算に比べて、ハノイの塔の計算の方が処理経過時間の増加が大きくなっている。これはハノイの塔の計算の方が、GC 処理が多く発生しているためと考えられる。

mutator の停止時間

fork システムコールによる GC プロセス生成時の mutator の停止時間を表 4（マルチ CPU マシン）および表 5（シングル CPU マシン）に示す。

停止時間の計測には、現在時刻を取得する gettimeofday 関数を用いている。ForkGC における mutator の停止時間は、fork システムコールを実行してプロセスを生成する間の時間、停止-回収型 GC の場合は､印付けと回収にかかる GC 全体の時間で、単位はミリ秒である。各ベンチマークプログラムに対して 10 回ずつ実行し、ベンチマークプログラムの実行中に行われた各 GC サイクルにおける mutator の停止時間を計測した。

表 4. Mutator の停止時間(multiCPUs).

GC の種類	boyer (ms)	ack (ms)	hanoi (ms)
パイプ型	13	13	13
共有メモリ型	13	13	13
停止-回収型	60～100	60	60

表 5. Mutator の停止時間(singleCPU).

GC の種類	boyer (ms)	ack (ms)	hanoi (ms)
パイプ型	6～245	7～467	6～485
共有メモリ型	6～246	7～203	6～232
停止-回収型	45～78	42～46	40～50

マルチ CPU マシン上での mutator の停止時間は、パイプ型、共有メモリ型の両方の SB 型 Fork GC において一定となっている。この停止時間は、平均の時間であるが、実際にはほぼ一定で、標準偏差は約 0.5 ミリ秒である。停止-回収型マークスイープ GC

における boyer ベンチマークの計算で、mutator の停止時間が変化しているのは、生存セル量が増え、印付けにかかる時間が増加していくためである。

シングル CPU マシン上での Fork GC における mutator の停止時間はかなり変化している。これは、単一プロセッサによる疑似並列処理のためである。fork システムコールの実行後、GC プロセスや全く別のプロセスがディスパッチされる場合があるので、mutator の停止時間は大きく変化する。

通常、Solaris におけるユーザプロセスのスケジューリング方法は、タイムシェアリング方式である。タイムスライスは優先度によって変化するが、20 ミリ秒から 200 ミリ秒である。

そのため、mutator が 200 ミリ秒以上停止しているのは、fork システムコールの実行後に、mutator プロセスとは別のプロセスに対してプロセッサが割り当てられているためと考えられる。

Fork GC の場合は、COW におけるコピー動作や、パイプ型 Fork GC の場合の、スイープ時のゴミセルの追加においても mutator の停止が発生するので、この評価だけでは完全な停止時間の評価とはいえない。この評価の範囲では、マルチ CPU マシンにおいての一回あたりの mutator の平均停止時間は、Fork GC の方が短くなっている。

ヒープサイズと mutator の停止時間

マルチ CPU マシン上での、ヒープサイズを変化させた時の、fork システムコールによる mutator の平均停止時間を図 6 に示す。横軸が、ヒープのセルサイズ（セル数）で、縦軸が平均停止時間（ミリ秒）である。停止時間の計測には、パイプ型の Fork GC を使用している。

ヒープサイズを増やすにつれて、mutator の停止時間が線形で増加している。これは、停止時間が、プロセス生成時における仮想メモリ空間のコピーコストに依存するためと考えられる。

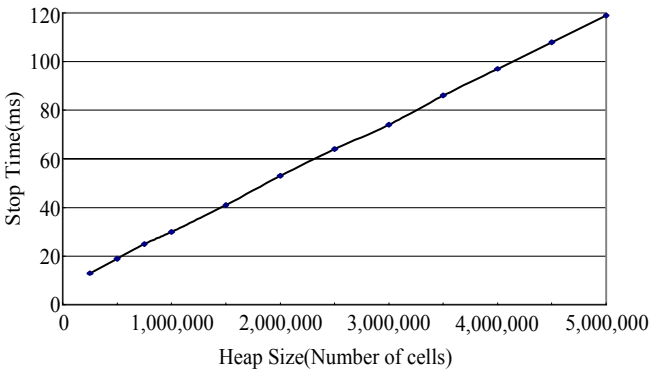

図 6. ヒープサイズと停止時間(multi CPUs).

Fork GC のバリアの問題点

処理時間全体の評価において Fork GC の場合の処理時間が、停止-回収型マークスイープ GC の場合よりも長くなった原因について考察する。

通常、純粋な SB 型並列 GC の場合には、必要なバリアは書き込みバリアだけで、読み込みや生成のバリアは発生しない。しかし Fork GC の場合には、GC の処理中に mutator が cons 関数でセルを生成するとセルが書き換えられ、mutator から GC への COW によるコピー処理が発生する。そのため、書き込みバリアだけではなく生成バリアも発生する。

一般的に実行頻度は、読み込み>生成>>書き込みなので [8]、GC の処理中に mutator によるセルの生成が頻繁に発生し、生成バリアによるコピー処理によって、処理時間が増加していると考えられる。COW によるコピー処理は、カーネルレベルで処理されるため、ユーザ時間よりもシステム時間の方が大幅に増加している。

Fork GC の場合、純粋な SB 型並列 GC のようにセル生成時のバリアをなくすことはできない。したがって、バリアによるオーバーヘッドは、書き込み頻度だけではなく、セル生成の頻度にも依存することになる。Rodrigues-Rivera らが対象とした一般アプリケーションのように、メモリ要求におけるサイズが大きく頻度が小さい場合には大きな問題とならないが、Lisp のように、サイズが小さく頻度が大きい場合には大きなオーバヘッドとなり、システム全体の処理能力へ影響する。バリアのオーバヘッドに関しては、この Fork GC は、cons 頻度が小さく、GC のマーキング処理量の小さなアプリケーションでないかぎり、SB 型並列 GC に大きく劣ると考えられる。

結論

fork システムコールを用いることで、Lisp 処理系に実装された一般的な停止-回収型マークスイープ GC を容易に並列型の GC へと変更することができた。シングル CPU マシンにおいても疑似並列処理により動作させることができた。マルチ CPU マシン上での、GC プロセス生成時における mutator の停止時間においては、停止-回収型マークスイープ GC の停止時間よりも短縮することができた。しかし、fork システムコールによるオーバーヘッドは予想以上に大きかったために、Fork GC の処理時間は、停止-回収型の 2 倍から 3 倍となった。

COW におけるコピー処理のオーバーヘッドを減らすためには、Fork GC の処理時間を短くする必要がある。つまり、Fork GC のマーキング処理量を減

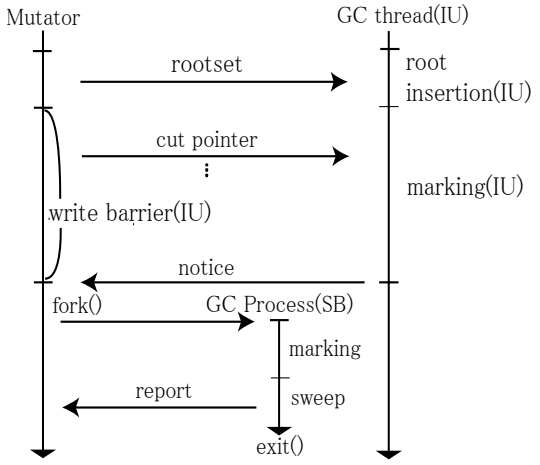

図 7. 相補型 Fork GC.

らせばよい。そのために今後は、相補型 (Complementary) GC[5,7] に Fork GC を組み込むことを計画している。

Fork GC を用いた相補型 GC を図 7 に示す。相補型 GC は、IU 型アルゴリズムのルートセット走査と追加の印付けを SB 型アルゴリズムで行う。IU 型でルート挿入と印付けを行った後に、GC プロセスを生成して追加の印付けと回収を行う。IU 型のルート挿入から GC プロセスによる回収までが 1 つの GC サイクルとなる。

IU 型アルゴリズムの問題点は、mutator を停止した状態でルートセットの走査と追加の印付けを行わなければならない点である。この追加の印付けを mutator と並列に実行できれば、mutator の停止時間を予測することが可能になる。

後半の Fork GC による SB 型の印付けは、前半の IU 型の印付け中に書き換えられたセルのための追加の印付けなので、Fork GC の処理時間は短くなる。また、通常の相補型 GC では IU 型と SB 型の両方の書き込みバリアが必要となるが、Fork GC を用いることで COW におけるコピー処理により、SB 型の書き込みバリアが不要となる。

文献

1) Gabriel RP (1985) *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge.
2) Jones R and Lins RD (1996) *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Chichester.
3) 佐藤憲一郎, 松井祥悟 (2004) fork システムコールを用いた並列ガーベジコレクション. *情報処理学会第 66 回全国大会講演論文集 (1).* pp.309-310.
4) 藤森 誠 (2002)プロセス生成型ガーベジコレクションの研究. *2001 年度神奈川大学理学部松井研究室卒業研究論文.*
5) Matsui S, Tanaka Y, Maeda A and Nakanishi M (1995) Complementary garbage collector. In: *Proc. Int. Workshop on Memory Management, Lecture Notesin Computer Science Vol.986.* Springer-Verlag, Kinross, Scotland. pp.163-177.
6) Rodriguez-Rivera G and Russo V (1997) Non-intrusivecloning garbage collection with stock operating system support. *SPE.* **27**: 885-904.
7) 松井祥悟，田中良夫，前田敦司，中西正和(1995)相補型ガーベジコレクタ. *情処学論* 36: 1874-1884.
8) 松井祥悟，田中良夫，前田敦司，中西正和(1999)並列ガーベジコレクションの実用化技術. *第 40 回プログラミング・シンポジウム報告集.* pp.159-170.
9) Wilson PR (1992) Uniprocessor garbage collection techniques. In: *Proc. Int. Workshopon Memory Management, Lecture Notesin Computer Science No.637.* Springer-Verlag, St.Malo, France. pp.17-32.
10) 田中良夫，松井祥悟，前田敦司，中西正和(1995)部分印付けを併用した並列 GC の提案および効率の解析. *信学論(DI)* J78-D-I: 926-935.