Ruby 1.9での高速なFiberの実装

芝　哲史[†]　笹田　耕一[†]

本発表では，Ruby 1.9 が提供する Fiber という機能の高速な実装を報告する．Fiber とは，プログラマに対して並行処理をサポートするための機構であり，状態を持つ処理を並行に扱う際に有用である．しかし，Ruby1.9 における Fiber は速度面に問題がある．そこで我々は，Ruby 1.9 処理系に対して Fiber の高速化を行い，複数の環境で評価を行った．その結果，Fiber の速度の向上を確認することができた．本発表では，Ruby 1.9 の現在の実装と我々の行った実装の紹介と，その性能評価を行う． そして，それらが実際に Fiber を利用する Ruby プログラムに対して，どのような影響を与えるかを考察する．

A Fast Fiber Implementation for Ruby 1.9

Satoshi Shiba[†] and Koichi Sasada

In this presentation we show a fast Fiber implementation for Ruby 1.9. Fiber supports concurrent programming and is useful for concurrent processing with state. However, the implementation of Fiber in Ruby 1.9 has performance problem. We implemented a faster Fiber for Ruby 1.9 and evaluated it on various environments. As a result, our proposed implementation is faster than the current one in most case. In this presentation we evaluate the ability for current Fiber implementation and proposed Fiber implementation. We also evaluated the effect of faster fiber implementation on practical Ruby application using Fiber.

1. はじめに

世界中で広く利用されているプログラミング言語 Ruby[1] は，2007 年に Ruby 1.8 から Ruby 1.9 へとバージョンアップし，それに伴う変更点の 1 つとして，並行処理をサポートする Fiber というクラスが, 組み込みクラスに新しく追加された．

Ruby 1.9 の Fiber クラスは一般にはコルーチン[6] やセミコルーチンと呼ばれている機構を実現する．これは, ノンプリエンプティブなユーザレベルスレッドとも見ることができる．この機構は，実行中の処理をプログラマが明示的に停止し，別の停止中の処理を再開することで，複数ある処理を並行に実行することを可能とする．本稿では，現在実行している Fiber を停止して，別の Fiber を再開することを，**Fiber 間のコンテキストスイッチ**と呼ぶ．

図 1 に Ruby 1.9 における Fiber クラスの利用例を示す．図 1 では，1 から 100 までの各値を 1 つずつ取り出すという処理を，Fiber を用いて記述している．

図 1 では，まず，Fiber.new にて Fiber を新しく生成する．このとき引数として渡したコードブロックが，Fiber を利用した並行処理の対象となる．生成した Fiber を用いた並行処理は，Fiber のインスタンスメソッドである `Fiber#resume` によって Fiber の持つコードブロックに処理を移し (1)，クラスメソッドである `Fiber.yield` によって，現在実行中の処理を中断し，`Fiber#resume` を呼び出した箇所へ実行を再開する (2) ことによって行う．

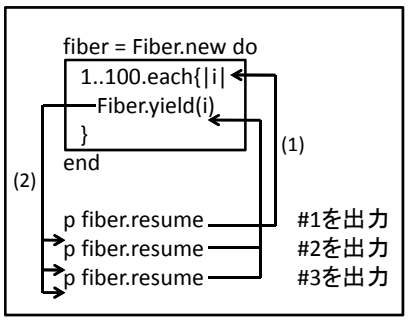

図 1　Fiber の利用例

Fiber は様々な場面で有用である．例えば，繰り返しを一般化する機構である Enumerator という組み込みクラスの `Enumerator#next` というメソッドは，Fiber を用いて実装されている[2]．

また，Fiber は `Enumerator#next` メソッドのような機能を実現する以外にも，ゲームプログラミングに

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

```
#Fiber を利用しない場合
def movecharactor(c)
  case c.status
  when INITIALIZE
    #初期化処理
    c.status = STATUS_1
  ...
  when STATUS_N
    c.status = FINALIZE
  when FINALIZE
    #終了処理
end

#Fiber を利用した場合
charactor = Fiber.new do |c|
  #初期化処理
  ...
  #終了処理
end
```

図 2 ゲームプログラミングにおける Fiber の利用例

おける各キャラクタなどのオブジェクトに対する処理や，イベントドリブンのプログラミングにおけるループ処理を記述する場合に有用である[7]．

ゲームプログラミングの場合，ゲーム内のキャラクタなどのオブジェクトがそれぞれ個別の状態を持ち，ゲームの進行に応じて，オブジェクトの状態は変化していく．そのため，オブジェクトの挙動を記述するメソッドの中では，図 2 に示す `movecharactor` メソッドのようにオブジェクトの状態に応じて処理を変化させる必要がある．このような状態に応じた条件分岐は簡潔に記述することは難しいが，図 2 のように Fiber を用いることで，状態を変数の中だけでなく，コード上の位置によって保持することができるため，処理を直感的に記述することができる．

イベントドリブンのプログラミングの場合，通常，イベント発生前の処理とイベント発生後の処理を連続的に記述することができないが，イベントハンドラの登録後に Fiber を中断し，イベントハンドラの中で Fiber の再開をすることで，イベント発生前とイベント発生後の処理を連続的に記述することができる．Fiber を用いることで，実行フローが複雑になりがちなイベント処理の見通しをよくすることができる．

Fiber クラスを利用して並行処理を行う場合，Fiber の生成コスト，Fiber の終了コストと，Fiber 間のコンテキストスイッチのコストが新しく発生する．そのため，Fiber によるパフォーマンスの低下を防ぐために，これらのコストを抑えることが，Fiber クラスを実装するうえでの課題となる．

しかし，Ruby 1.9 においては，これらの新しく発生するコストのうち，Fiber 間のコンテキストスイッチのコストが高い．Fiber を利用するスクリプトにおいて，これらが処理全体のボトルネックとなる場合もある．

そこで，我々はこの問題を解決するために，Ruby 1.9 に対して OS などのシステムが提供するコンテキストスイッチのための仕組みを利用して，高速な Fiber 間のコンテキストスイッチを実装し，評価した．その結果，我々の実装では，Ruby 1.9 の Fiber よりも高速に動作することが確認できた．ただし，最大生成数が Ruby 1.9 の Fiber に比べて減少する．

本稿の構成を以下に示す．まず，2 章で Ruby 1.9 における Fiber の生成，終了，Fiber 間のコンテキストスイッチという Fiber のプリミティブな処理の実装，そのボトルネックについて分析する．次に，3 章で Fiber のボトルネックを解決するための手法を検討する．4 章で検討した手法に対する実装について述べ，5 章で評価を行い，6 章でまとめる．

2. Ruby 1.9 における Fiber の実装

Fiber はそれぞれ Ruby スクリプトの処理に必要となる固有のコンテキスト持つ．Fiber が固有に持つコンテキストとしては，Ruby 処理系内の仮想マシンレベルのスタックやレジスタ，ハードウェアレベルのスタックやレジスタが挙げられる．以降，ハードウェアレベルのスタックのことを，**マシンスタック**，ハードウェアレベルのレジスタのことを**マシンレジスタ**と呼ぶ．

Fiber の生成，Fiber 間のコンテキストスイッチ，Fiber の終了においては，これらのコンテキストを適切に割り当て，切り替え，解放する処理が行われる．

本章では，Fiber の生成，Fiber 間のコンテキストスイッチ，Fiber の終了という Fiber のプリミティブな処理について，Ruby 1.9 処理系での実装を述べ，Fiber の持つボトルネックを分析する．

2.1 Fiber の生成

Fiber はそれぞれが，Ruby スクリプトの処理に必要な実行コンテキストを固有に持つ．そのため，Fiber を生成するためには，これらの実行コンテキストを割り当てる必要がある．

Fiber の実行コンテキストのうち，マシンスタック領域以外は，Fiber の生成時に割り当てられる．マシンスタック領域は，生成された Fiber に初めてコンテキストスイッチを行ったときに割り当てる．この工夫により，生成されたが使われない Fiber へのマシンスタック領域の割り当てを回避することができる．

2.2 Fiber の終了

Fiber に渡されたコードブロックの実行が全て終わると，Fiber は終了処理を行う．Fiber の終了処理では，終了した Fiber がスレッドの持つルートの Fiber だった場合，そのスレッドを終了する．そうでない場合は，終了した Fiber の親に対してコンテキストスイッチを行う．

Fiber の終了処理の時点で，Fiber の実行コンテキストを解放することができるが，Ruby 1.9 の現在の実装では Fiber の終了処理での実行コンテキストの解放は行わない．Fiber の実行コンテキストの解放は

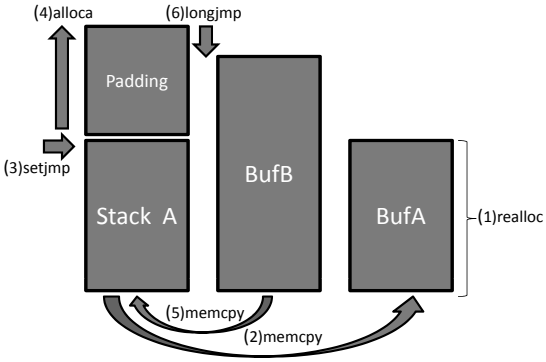

図 3 マシンスタックの切り替え

GC でのスイープ処理にて行う．

2.3 Fiber 間のコンテキストスイッチ

ある Fiber から別の Fiber へと切り替える場合は，実行中の Fiber のコンテキストを退避し，切り替え先の Fiber のコンテキストを復元する．

VM のコンテキストの切り替えは，Ruby 1.9 処理系内部で VM を表すデータ構造が持つポインタの値を書き換えることによって行う．また，ハードウェアのコンテキストの切り替えは，マシンスタックに対してコピー処理を行い，マシンレジスタを `longjmp` によって書き換えることによって行う．

このように，VM のコンテキストの切り替えはポインタの書き換えのみで完了するが，ハードウェアのコンテキスト切り替えには，マシンスタックのコピー処理というコストの高い処理が行われる．

図 3 に Ruby 1.9 において，Fiber A から Fiber B へとコンテキストスイッチを行う際の，マシンスタック切り替えの様子を示す．図 3 では，まず Fiber A のマシンスタック Stack A を退避するためのバッファである Buf A のための領域を `(1)realloc` で準備し，`(2)memcpy` にて Stack A を Buf A へとコピーし，`(3)setjmp` でマシンレジスタを保存する．次に，Stack A のサイズが Fiber B のマシンスタック Stack B のサイズよりも小さい場合は `(4)alloca` や再起呼び出しを行うことでマシンスタックを拡張する．そして，B の内容を `(5)memcpy` にて復元し，最後に `(6)longjmp` にて，マシンスタックのトップを指すマシンレジスタであるスタックポインタの切り替えを行い，Fiber B のマシンスタックの復帰を完了する．

このように，Ruby 1.9 処理系ではマシンスタックの切り替え処理では複雑な操作を行う．しかし，マシンスタックの切り替えを行うならば，マシンスタック用領域を切り替え元と切り替え先で別々に取り，スタックポインタを切り替えるだけで済む．つまり，マシンスタックの切り替えは，上記のような複雑な処理を行わなくても可能である．

それにもかかわらず，マシンスタックの切り替えの際に，図 3 のような処理を行うのは，Ruby 1.9 の移植性を高めるためである．

Ruby 1.9 処理系がスタックポインタの退避，復元に用いる `set/longjmp` は，`setjmp` で引数として与えたバッファにマシンレジスタの値を保存し，`longjmp` で保存したマシンレジスタの値を復元する．このとき，`longjmp` では，引数である `jmpbuf` を直接操作しない限り， `setjmp` を呼んだとき以外の値にマシンレジスタの値を設定することができない．Fiber に対して新しいマシンスタック領域を割り当てるには，`longjmp` の扱う引数である `jmpbuf` の直接操作や，マシンレジスタの直接操作を行う必要がある．

しかし，アセンブリ言語や `jmpbuf` は環境に依存するため，これらを直接操作することは移植性を損なう．そのため，Ruby 1.9 では Fiber は固有のマシンスタック領域を持たず，1 つのマシンスタック領域を共有している．この共有処理のために，マシンスタックのコピーがマシンスタック切り替えの際に必要となる．

そのため，Ruby 1.9 では，Fiber 間コンテキストスイッチの際に，マシンスタックのサイズに比例したコストが発生する．Fiber 間のコンテキストスイッチにおいては，マシンスタックの切り替え処理がボトルネックとなっている．

2.4 Fiber のボトルネック

Fiber の生成，終了，Fiber 間のコンテキストスイッチでは，それぞれ重い処理が行われるが，Fiber の生成，終了処理に関しては各 Fiber に対して最大で 1 度しか行われない．しかし，Fiber 間のコンテキストスイッチに関しては，通常，各 Fiber に対して複数回呼ばれる．`Fiber#resume`，`Fiber.yield` を用いて Fiber 間を行き来するだけで，2 度のコンテキストスイッチが行われる．

このため，Fiber のボトルネックは，Fiber 間のコンテキストスイッチのボトルネックである，マシンスタックの切り替え処理にある．

3. Fiber の高速化手法の検討と設計

Fiber のボトルネックは，Fiber 間コンテキストスイッチでの，マシンスタックの切り替えにある．このマシンスタック切り替え処理が遅いのは，マシンスタック用のメモリ領域を共有しているために，共有領域の退避，復元処理が必要となるためである．以降，単に領域と書いた場合，マシンスタック用のメモリ領域のことを指すものとする．

マシンスタック用領域を共有する場合，この退避，復元処理は避けられないため，マシンスタック切り替えの高速化を行うには領域を別に確保する必要がある．そのため，領域を新しく割り当てるための手法，領域を解放するための手法と，新しく確保した領域に対するマシンスタック切り替えの手法が必要となる．また，

Ruby の移植性を保つために，これらの手法は移植性を考慮したものである必要がある．

本章では，新しいマシンスタックで Fiber を動作させるための既存手法についてまず述べ，マシンスタック切り替えの手法と，領域の割り当て，解放の手法について検討する．

3.1 既存手法

コルーチンの機能を Perl に追加するライブラリ Coro[3] では，固有のマシンスタック領域を持つコルーチンを実現するために，以下に示す手法を用いている．

(1) `jmpbuf`(`set/longjmp` の引数) の直接操作
(2) `get/set/makecontext` 関数の利用
(3) `sigaltstack` 関数の利用

`set/longjmp` の引数である `jmpbuf` を操作すれば，`longjmp` でマシンレジスタの値を任意に設定することができるため，固有のマシンスタック領域を持つ Fiber を実現することができる．この手法は高速だが，`jmpbuf` が環境依存であるため，移植性が低い．

`get/set/makecontext` はコンテキストを操作するための POSIX API[5] で，`get/makecontext` が `setjmp` に，`setcontext` が `longjmp` に対応する．これらの API は `set/longjmp` とは違い，切り替わるコンテキストのエントリポイントとマシンスタック領域を指定することができるため，新しいマシンスタック領域で Fiber を動作させることができる．また，この API は Windows 環境ではサポートされていないが，POSIX 環境での移植性は高い．しかし，`get/setcontext` は内部でシグナルマスクの設定を行う `sigprocmask` というシステムコールを呼び出すため，`set/longjmp` と比べてオーバーヘッドが大きい．

`sigaltstack` は，シグナルハンドラの利用するマシンスタックを登録するための API である．`sigaltstack` を利用する手法では，`sigaltstack` によってシグナルハンドラの利用するマシンスタック領域を登録し，シグナルを発生させ，シグナルハンドラ内で `setjmp` を呼ぶことで，新しいマシンスタック領域に対する `longjmp` を行う．この手法も，`get/set/makecontext` と同様に，POSIX 環境での移植性は高い．しかし，この手法はシグナルハンドラの登録とシグナルの発生が必要になるため，初期化コストが `set/get/makecontext` よりも高い．

3.2 切り替え手法の検討

マシンスタック領域の切り替えの手法は，Windows 環境と POSIX 環境で利用可能な API が大きく異なるため，それぞれの環境で手法を別に用意する．

Windows 環境では `SwitchToFiber` という API を用いる．Windows は，Ruby 1.9 の Fiber に似た機能[4] を API として提供しているため，それをそのまま利用する．

POSIX 環境では，移植性と初期化コストを考慮し，`set/get/makecontext` と `set/longjmp` を用いる．Ruby の場合，スレッドをまたいだ Fiber 間のコンテキストスイッチを行うことは認めていない．また，Fiber 単位でシグナルマスクの設定を行うことも認めていない．そのため，同じスレッドに属する Fiber のシグナルマスクは全て等しい．そこで，`set/get/makecontext` で Fiber を 1 度新しいマシンスタック領域で動作させた後は，`sigprocmask` を呼ぶ `get/setcontext` よりも高速な，`set/longjmp` を用いてマシンスタック領域の切り替え処理を行う．

3.3 割り当て，解放手法の検討

マシンスタック領域の割り当て，解放の手法も，マシンスタック領域の切り替えの手法と同様に，Windows 環境と POSIX 環境で利用可能な API が大きく異なるため，それぞれの環境で手法を別に用意する．

Windows 環境では C 言語にて Ruby 1.9 の Fiber に似た機能を提供する API が存在するため，それをそのまま利用する．Windows 環境では，Fiber 用のマシンスタック領域の割り当て，解放には，`CreateFiberEX` と `DeleteFiber` という API を用いる．

POSIX 環境では，マシンスタックを新しく割り当てる場合，マシンスタックのオーバーフローを検知する必要があるため，マシンスタック領域の末尾にページ保護をかけ，スタックオーバーフローを検知する．ページ保護の設定には，`mprotect` を用いる．

このとき，`mprotect` による保護の単位はページ単位であるため，マシンスタック用領域の割り当て，解放には，`mmap/munmap` を用いる．

また，スタックオーバーフローが起こったときにマシンスタック領域の拡張が行えるとは限らないため，マシンスタック領域は，Fiber の生成時にまとめて確保する．

4. 実　　装

前章で検討した結果を基に，Fiber の高速化のための実装を行った．

検討した手法では Fiber ごとに必要となるマシンスタック領域の割り当てを行うたびに mmap，mprotect の 2 回のシステムコールを呼ぶ．このコストを抑えるために，マシンスタック領域を一定本数キャッシュすることにした．今回はキャッシュするマシンスタックの本数は 10 とし，Fiber の生成時に確保するマシンスタック領域のサイズは 64KB とした．この 64KB のうち，末端の 4KB を `mprotect` によるスタックオーバーフローの検出に用いるため，実際に Fiber が利用できるマシンスタック領域のサイズは 60KB となる．

また，検討した手法の移植性を確認するために，複数の環境に対して実装を試みた．今回試みた実装の対象を表 1 に示す．

今回検討した手法の実装対象としたのは，windows 環境と，`set/get/makecontext` をサポートしている

表 1　実装対象

OS	ビット数	テスト環境
Windows	32	実機上
Linux	32	実機上
Linux	64	実機上
Solaris	32	実機上
MacOSX	32	実機上
OpenSolaris	32	VMM 上
FreeBSD	32	VMM 上
DragonFlyBSD	32	VMM 上

環境のみとした．これ以外の環境では，従来どおりの実装が利用できる．また，実機でのテスト環境を構築できない場合は，VMware 社の提供する仮想マシンモニタ (VMM) である VMware Player 上でテストを行った．

表 1 に示す以外にも，OpenBSD，NetBSD という POSIX 準拠の広く利用されている OS があるが，今回この 2 つの OS は以下の理由で実装の対象外とした．

NetBSD は `set/get/makecontext` をサポートしているが，NetBSD の pthread では，スレッドローカルな領域がマシンスタックの位置に依存しているため，pthread 環境下で Fiber ごとに個別のマシンスタック領域を持つことができない．例えば，スレッド間の排他制御で必要となるスレッドの識別子は，マシンスタックのアドレスに依存しているため，マシンスタック切り替えの前に取得したロックを，マシンスタック切り替えの後に解放することが出来ないという事態に陥る．このため，ネイティブスレッドを利用している Ruby 1.9 では，NetBSD において検討した手法を用いることができないため，NetBSD は実装の対象外とした．

OpenBSD は `set/get/makecontext` をサポートしていないことから，実装の対象外とした．

5. 評　　価

本章では現在の Ruby 1.9 処理系の実装と，我々の行った実装を比較するための Fiber の基本性能評価と，IO 多重化に Fiber を用いた場合の評価を行う．

以降，我々の行った Fiber の実装を proposed と，Ruby 1.9 処理系の現在の Fiber の実装を current と呼ぶ．

今回の行った Fiber の性能評価には，本稿の執筆時点で最新の安定版としてリリースされている `Ruby1.9.1-p243` を用いた．

また，基本性能評価では表 2 に示す複数の OS，アーキテクチャを用いた．IO 多重化に Fiber を用いた場合の評価では，表 2 に示す Linux32 環境でサーバプログラムを動作させた．

5.1　基本性能評価

current と proposed の間で Fiber の基本性能を比較するための評価項目を以下に記す．

表 2　評価環境

環境	OS	CPU	コンパイラ
Windows	Windows7 32-bit	IntelCore2Duo 2.00GHz	VC2008
Linux32	Linux 2.6.31 32-bit	IntelCore2Quad 2.66GHz	GCC4.3.3
Linux64	Linux 2.6.26 64-bit	IntelXeon 2.00GHz	GCC4.3.2
Solaris	SunOS 5.10 32-bit	spqrcv9 1.167GHz	GCC3.4.3
MaxOSX	MacOSX 10.5.8 32-bit	IntelCore2Duo 2.20GHz	GCC4.0.1

表 3　times メソッドによるマシンスタックの拡張サイズ

環境	拡張されるサイズ (Byte)
Windows	536
Linux32	2048
Linux64	1456
Solaris	1144
MacOSX	2384

- コンテキストスイッチの速度
- 生成，解放速度
- 生成数

本節では，これらの評価項目それぞれについての評価を行う．

5.1.1　コンテキストスイッチの速度

Fiber 間コンテキストスイッチの速度の評価には，図 4 に示すベンチマークを用いた．図 5 にその実行結果を示す．

図 4 では，Fiber を 1 本生成し，RECURSIVE 回の再起呼び出しを行うことでマシンスタックを伸ばし，COUNT 回 Fiber 間のコンテキストスイッチを行う．

Ruby 1.9 処理系では，メソッドの再起呼び出しで拡張されるのはマシンスタックではなく VM のスタックであるため，再起呼び出しの際にはマシンスタックの拡張を伴う `times` メソッドを呼び出す．再起呼び出しの回数が 1 増えるごとに，表 3 に示すサイズだけ，マシンスタックが拡張される．

図 5 より，評価を行った全ての環境において，Fiber 間のコンテキストスイッチの速度では，proposed は current よりも高速だということが分かる．

また，マシンスタックのサイズが増すほど，proposed と current の速度比が線形に増していくことが分かる．表 3 に示す `times` メソッドによるマシンスタックサイズの増加量が大きい環境ほど，再起呼び出しの回数の影響が顕著に出ている．これは，current ではマシンスタックのサイズが増すほど，Fiber 間のコンテキストスイッチの際のコピー処理のコストが増すためである．

5.1.2　生成，終了速度

Fiber の生成，終了速度の評価には，図 6 に示すベンチマークを用いる．図 7 にその結果を示す．

```
RECURSIVE = 0   #再起呼び出しの回数
COUNT = 3000000 #Fiber 間コンテキストスイッチの回数

def nest(i)
  if i == 0 then
    while true do
      Fiber.yield
    end
  else
    1.times do
      nest(i-1)
    end
  end
end

fiber = Fiber.new do
  nest(RECURSIVE)
end

COUNT.times do
  fiber.resume
end
```

図 4 Fiber 間コンテキストスイッチのベンチマーク

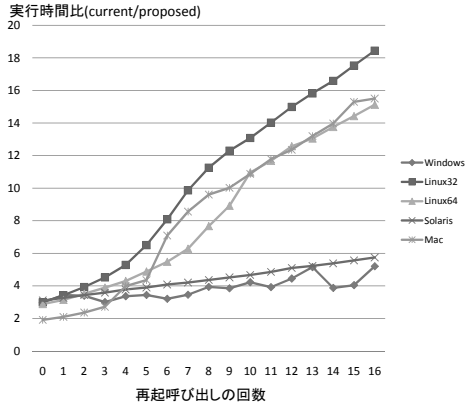

図 5 Fiber 間コンテキストスイッチのベンチマーク結果

図 6 では，Count 回，ループ A とループ B を処理する．ループ A では Size 本の Fiber を 1 度に生成し，ループ B では Size 本の Fiber をまとめて処理する．このスクリプト全体では，Count * Size 回の Fiber の生成，終了処理が呼ばれる．

図 7 より，Windows 以外の環境では Size が 10 を越えた時点で，proposed の速度が低下していることが分かる．これは，proposed ではマシンスタックのキャッシュの本数を 10 本としているため，キャッシュをしている本数よりも多くの Fiber を一度に生成した場合に，`mmap`，`mprotect`，`munmap` といった，システムコールのコストが発生するためである．Windows ではマシンスタックのキャッシュ処理を行っていないため，一度に生成する Fiber の本数はほとんど影響しない．

また，current では Fiber 用のマシンスタック領域確保には GC のトリガとなる `ALLOC_N` という API を利用しているが，proposed ではマシンスタック領域の確保に `mmap` を用いており，これは GC のトリガとならない．そのため，current は proposed よりも Fiber 生成時における GC の発生回数が若干多い．

```
SIZE = 1                #まとめて処理する Fiber の本数
COUNT = 1000000 / SIZE #ループ A, B を実行する回数

a = Array.new(SIZE)
COUNT.times do
  SIZE.times do |i|    #ループ A
    a[i] = Fiber.new{}
  end
  SIZE.times do |i|    #ループ B
    a[i].resume
  end
end
```

図 6 生成，終了速度のベンチマーク

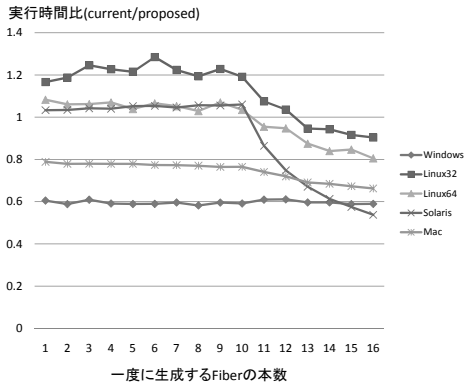

図 7 生成，終了速度のベンチマーク結果

GC の影響も考慮すると，生成，終了処理においては，マシンスタックのキャッシュが十分な Linux や Solaris 環境では proposed と current は同程度の速度，それ以外の環境では proposed が current よりも遅い．

5.1.3 生 成 数

Fiber の生成数の評価には，図 8 に示すベンチマークを用いる．図 9 にその結果を示す．

図 8 では，無限ループの中で Fiber の生成のみを行う．生成した Fiber が GC 時にスイープされないように，スクリプトの開始行で GC を禁止している．

図 8 で生成するオブジェクトは，実行コンテキストが最小限の Fiber オブジェクトがほとんどであるため，このベンチマークの結果は，ベンチマークを動作させた環境下での，Fiber の最大生成数とほぼ一致する．

図 9 より，proposed における最大生成数は，32bit 環境と 64bit 環境で大きく異なる．32bit 環境では，proposed における Fiber の生成数は，プロセスがユーザに対して提供しているアドレス空間のサイズをマシンスタック領域のサイズ（今回は 64KB）で割った量に近い．例えば，32bit の Linux がプロセスに対して提供するアドレス空間のサイズは 3GB であるため，Linux32 では Fiber の生成数が 4 万から 5 万ほどに収まっている．proposed の場合 32bit 環境では仮想メモリが Fiber の生成数の制限となっている．

```
GC.disable
i = 0
while true do
  fiber = Fiber.new do
    Fiber.yield
  end
  fiber.resume
  p i += 1
end
```

図 8 最大生成数のベンチマーク

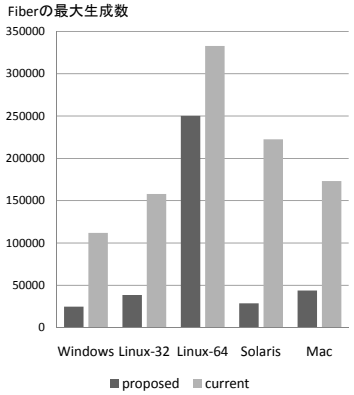

図 9 最大生成数のベンチマーク結果

それに対し，64bit の Linux がプロセスに対して提供しているアドレス空間のサイズは十分であるため，proposed での Fiber の最大生成数が大幅に増加している．しかし，proposed で各 Fiber の持つマシンスタック領域に割り当てられる実メモリは，システムにおけるページサイズ単位での割り当てになるため，64bit 環境においても current に比べて生成可能な Fiber の数は少ない．

5.2 Fiber を用いた IO 多重化における評価

IO 多重化におけるイベントループ処理は，Fiber が有用な場面の 1 つである．本節では，Fiber を用いた IO 多重化における，proposed と current の速度比を評価する．評価には図 11 に示すベンチマークを用いる．図 10 にその結果を示す．

図 11 では，コネクションを保ったまま，複数回の通信を行う処理をシミュレートする．まずコネクションを CONNECTION 個確立し，メインループの中で 100 万回のエコー処理を行う．メインループの中では，ソケットを `select` で多重化し，その結果を契機とするイベントドリブンの処理を行う．IO の多重化処理をイベントループで行う場合，実行フローが複雑になりやすいが，Fiber を用いることで，見かけ上同期的な IO 命令で，`select` によるノンブロッキングな IO 処理を利用することができる．

図 10 より，再起呼び出しを行わない場合は proposed の速度が 20%ほど current を上回っている．再起呼び出しによって，マシンスタックのサイズが大きくなると，proposed と current の速度差も線形に大きくなる．このように proposed では，current に比べて，IO 多重化を用いたサーバ処理をより高速に行うことができる．

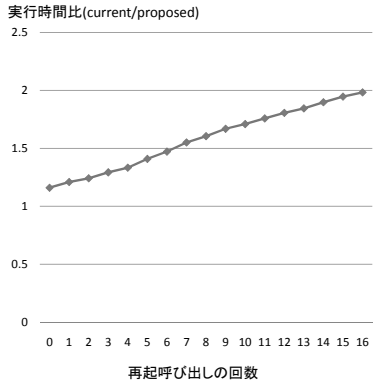

図 10 Fiber を用いた IO 多重化のベンチマーク結果

5.3 考　　察

Fiber 間コンテキストスイッチの速度では，評価を行った全ての環境で proposed が current を上回り，Fiber の生成，終了速度では多くの場合，proposed が current を下回ることが確認できた．しかし，通常，Fiber の生成，終了処理が行われる回数よりも Fiber 間のコンテキストスイッチ行われる回数の方が多いため，実際に Fiber を利用するアプリケーションでは多くの場合に，proposed が current よりも高速に動作することが期待できる．

Fiber の最大生成数では，評価を行った全ての環境で proposed が current よりも少ない．これは，Fiber の利用する最大サイズのアドレス区間を Fiber の生成時に割り当てることから，32bit の環境では，プロセスのアドレス空間のサイズが制限となるためである．64bit の環境では，アドレス空間のサイズは十分なため，proposed と current の差は縮まるが，proposed では各 Fiber が個別のマシンスタック領域を持つため，マシンスタックのために使用する物理メモリはページサイズ単位になる．そのため，64bit 環境においても，既存手法に比べて Fiber の最大生成数は少ない．しかし，Fiber を実際に利用するアプリケーションにおいて，Fiber を万単位で生成することは稀であるため，Fiber の最大生成数が問題となることは少ない．

6. ま と め

本稿では，Ruby 1.9 の Fiber を高速化手法を検討，実装し，評価を行った．

まず，Ruby 1.9 における Fiber のプリミティブな処理について述べ，そのボトルネックが Fiber 間のコンテキストスイッチにおけるマシンスタックの切り替え処理であることを考察した．

次に，Fiber のボトルネックを解消するために，マシンスタック切り替えの高速化手法を検討した．検討した結果，システムが提供するコンテキストスイッチのための仕組みを利用することにした．Windows 環境の場合は Win32API が提供する `SwitchToFiber`，`CreateFiberEX`，`DeleteFiber` という API を，POSIX 環境の場合は POSIX API が提供する `get/set/makecontext`，`set/longjmp`，`mmap/munmap/mprotect` という API を利用する．

検討した手法を様々な環境に実装し，その移植性を確認した．さらに，ベンチマークを用いて，Fiber の基本性能を評価した．評価の結果，Ruby 1.9 の Fiber の実装に比べて，Fiber 間のコンテキストスイッチが 2 倍以上高速化することが確認できた．Fiber の生成，終了では Ruby 1.9 の Fiber の実装と比べてシステムコールのオーバーヘッドが発生するが，マシンスタックのキャッシュを用いて軽減できることが確認できた．Fiber の生成，終了処理が行われる回数よりも Fiber 間のコンテキストスイッチ行われる回数の方が一般的に多いため，Fiber を利用するアプリケーションでは多くの場合に，我々の実装の方が高速に動作する．我々の Fiber の実装の方が Ruby 1.9 の Fiber の実装に比べて Fiber の最大生成数が少ないことが確認できた．しかし，Fiber を実際に利用するアプリケーションにおいて，Fiber を万単位で生成することは稀であるため，Fiber の最大生成数が問題となることは少ない．

参 考 文 献

1) オブジェクト指向スクリプト言語 Ruby `http://www.ruby-lang.org/ja/`
2) オブジェクト指向スクリプト言語 Ruby リファレンスマニュアル `http://doc.okkez.net/static/191/doc/index.html`
3) libcoro `http://software.schmorp.de/pkg/libcoro.html`
4) MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More `http://msdn.microsoft.com`
5) IEEE POSIX Certification Authority `http://standards.ieee.org/regauth/posix/`
6) Conway, Melvin E. Design of a separable transition-diagram compiler. Commun. ACM, Vol.6, No.7, pp. 396-408(1963).
7) 長 慎也, 兼宗 進. ゲームプログラミングにおけるコルーチンの有用性. 情報処理学会論文誌. プログラミング, Vol49, No1, pp. 131, 20080115.


```
require 'socket'
require 'fiber'

CONNECTION = 64
FINISH = 1000000
RECURSIVE = 0

$fibers = {}
$sockets = {:read => [], :write => []}

def attach(socket, mode)
  $sockets[mode].push(socket)
  $fibers[socket.object_id] = Fiber.current
end

def detach(socket, mode)
  $sockets[mode].delete(socket)
  $fibers.delete(Fiber.current)
end

def mygets(socket)
  attach(socket, :read)
  Fiber.yield
  socket.gets
  detach(socket, :read)
end

def myputs(socket, s)
  attach(socket, :write)
  Fiber.yield
  socket.puts(s)
  detach(socket, :write)
end

def resume(ioset)
  ioset.each do |ioset|
    $fibers[ioset.object_id].resume
  end
end

def eventloop(recursive)
  if recursive == 0
    while true
      if result = IO.select($sockets[:read],
          $sockets[:write], nil)
        resume(result[0])
        resume(result[1])
      end
    end
  else
    1.times do
      eventloop(recursive - 1)
    end
  end
end

server = TCPServer.open(12345)
finish = 0
CONNECTION.times do
  Fiber.new do
    s = server.accept
    while true do
      str = mygets(s)
      myputs(s, str)
      finish += 1
      if finish == FINISH
        exit(0)
      end
    end
  end.resume
end

eventloop(RECURSIVE)
```

図 11 Fiber を用いた IO 多重化のベンチマーク