Vol. 48　No. SIG 0(PRO 34)　　情報処理学会論文誌：プログラミング　　2007

Ruby 用仮想マシン YARV における並列実行スレッドの実装

笹田　耕一 †1　松本行弘 †2
前田　敦司 †3　並木　美太郎 †4

本論文ではスクリプト言語 Ruby 用仮想マシン *YARV: Yet Another RubyVM* における並列実行スレッド処理機構の実装について述べる．Ruby はその使いやすさから世界中で広く利用されているプログラム言語である．Ruby の特徴のひとつにマルチスレッドプログラミングに対応しているという点があるが，現在広く利用されている Ruby 処理系は移植性を高めるため，すべてユーザレベルでスレッド制御を行っている．しかし，このスレッド実現手法では，実行がブロックしてしまう処理が C 言語レベルで記述できない，並列計算機において複数スレッドの並列実行による性能向上ができないなどの問題がある．そこで，現在筆者らが開発中の Ruby 処理系 YARV において，OS やライブラリなどによって提供されるネイティブスレッドを利用するスレッド処理機構を実装し，複数スレッドの並列実行を実現した．並列化にあたっては，適切な同期の追加が必要であるが，特に並列実行を考慮しない C 言語で記述した Ruby 用拡張ライブラリを安全に実行するための仕組みが必要であった．また，同期の回数を減らす工夫についても検討した．本論文では，これらの仕組みと実装についての詳細を述べ，スレッドの並列実行によって得られた性能向上について評価した結果を述べる．

An Implementation of Parallel Threads for YARV: Yet Another RubyVM

Koichi Sasada ,[†1] Yukihiro Matsumoto ,[†2] Atsushi Maeda [†3]
and Mitaro Namiki [†4]

In this paper, we describe an implementation of parallel threads for YARV: Yet Another RubyVM. The Ruby language is used worldwide because of its ease of use. Ruby also supports multi-threaded programming. The current Ruby interpreter controls all threads only in user-level to achieve high portability. However, this user-level implementation can not support blocking task and can not improve performance on parallel computers. To solve these problems, we implement parallel threads using native threads provided by systems software on *YARV: Yet Another RubyVM* what we are developing as another Ruby interpreter. To achieve parallel execution, correct synchronizations are needed. Especially, C extension libraries for Ruby which are implemented without consideration about parallel execution need a particular scheme for running in parallel. And we also try to reduce a number of times of synchronization. In this paper, we show implementations of these schemes and results of performance improvement on parallel threads execution.

1. は じ め に

スクリプト言語 Ruby[13),15),16)] は、手軽にオブジェクト指向プログラミングを実現するための種々の機能を持つプログラミング言語である．その使いやすさ[21)]から，Ruby は世界中で広く利用されており，多くのユーザを擁すプログラミング言語となっている．

Ruby の特徴の一つに，言語レベルでマルチスレッド実行に対応しているということが挙げられる．プログラミング言語におけるマルチスレッド機能とは，複数のスレッド（命令流）を並行，もしくは並列に実行する機能であり，最近の多くのプログラミング言語において標準でサポートされている．

Ruby において，スレッドを用いるプログラムは**図 1** のように記述される．Ruby プログラムにおいて生成されたスレッドを，以降 Ruby スレッドという．図 1 では，新しい Ruby スレッドを `Thread.new` によって生成する．`do ... end` で囲まれたブロックの中身 (A) とその後ろに書かれた処理 (B) は別スレッドとして

†1 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
†2 (株) ネットワーク応用通信研究所
Network Applied Communication Laboratory, Inc.
†3 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Infomation Engineering, University of Tsukuba
†4 東京農工大学大学院工学府
Graduate School of Technology, Tokyo University of Agriculture and Technology

```
m = Mutex.new
th = Thread.new do
  # (A) ここに記述した処理を新しいスレッドとして実行
  m.synchronize do
    # (Am) ここに記述した処理は (Bm) と排他に実行される
  end
end
# (B) ここに記述した処理は (A) と並行に実行される
m.synchronize do
  # (Bm) ここに記述した処理は (Am) と排他に実行される
end
# ...
th.join # ここでスレッド th と合流する
```

図 1 Ruby におけるスレッド生成と合流・排他制御の例
Fig. 1 Thread creation/joining/synchronization on Ruby Programming

並行に実行される．Ruby スレッドを Ruby プログラム中ではスレッドオブジェクトとして扱うことができ，図 1 では生成したスレッドオブジェクトを変数 `th` に代入している．スレッドオブジェクトに対して `Thread#join` メソッドを呼ぶことで，スレッドオブジェクトに対応する Ruby スレッドが合流するのを待つ．(Am)，(Bm) の処理は，プログラム冒頭で生成した Mutex オブジェクトにより排他制御される．

言語処理系にマルチスレッド機能を実現する方法として，ネイティブスレッド処理機構を利用する方式がある．ネイティブスレッド処理機構は OS など，システムが提供するスレッド管理機能であり，POSIX Thread（以降 Pthread）[5] などの仕様として提供されている．ネイティブスレッド処理機構が対応していれば，スレッドの論理的な並行実行だけではなく，並列計算機上での物理的な同時並列実行を行うことが可能である．なお，この機構により管理されるスレッドをネイティブスレッドという．

現在広く利用されている，松本が中心となって開発を行ってきた Ruby インタプリタ（以降，旧 Ruby 処理系）では，Ruby スレッドをユーザレベルスレッド☆として実現している[15),25)]．これは，複数の Ruby スレッドを一つのネイティブスレッド上で交互に実行させることで，論理的に並行実行するスレッドを実現する方式である．この方式は，スレッド生成などのスレッド制御が軽量に行えることや，スレッドスケジューリングを柔軟に行うことが可能であること，移植性の高さなどの利点がある．

しかし，ユーザレベルスレッドでは，常に一つのネイティブスレッドしか利用しないため，Ruby スレッドを並列計算機上で同時並列に実行することができない．近年のマルチコアプロセッサに代表される並列計算機のコモディティ化にともない，ソフトウェアの並列実行による高速化は今後ますます重要となる課題であるが，Ruby スレッドが並列に実行できないのは大きな問題である．

また，旧 Ruby 処理系によるユーザレベルスレッド実装では真の並行実行を実現出来ない．たとえば，ある Ruby スレッドが UNIX のシステムコールである select のような，実行をブロックしてしまう処理を実行すると，その他の Ruby スレッドへスケジューリングされずに全 Ruby スレッドがブロックしてしまう．旧 Ruby 処理系では実行をブロックする処理を行わず，処理系内部でポーリングを行いこの問題を回避しているが，いつもポーリングなどの代替手段を用いることが出来るとは限らない．

そこで，筆者らが開発している Ruby 処理系である *YARV: Yet Another RubyVM*[8),24)]（以降 YARV）に，ネイティブスレッド処理機構を利用して並列実行が可能な Ruby スレッドを実現した．YARV は Ruby 用バイトコード実行型仮想スタックマシンであり，Ruby プログラムを YARV 命令列にコンパイルして実行する．YARV は旧 Ruby 処理系よりも高速に動作する処理系を目指しており，さまざまな工夫を行っている．本研究では YARV に並行・並列実行を行うための Ruby スレッド処理機構を実現した．

並列実行する Ruby スレッドを実現するにはいくつか課題があるが，特に過去のプログラム資産を利用可能にすることは重要である．Ruby の特徴の一つに，世界中で開発されている多くの Ruby 用拡張ライブラリが利用可能という点がある．拡張ライブラリは C 言語などで実装されるネイティブメソッドによって構成されているが，旧 Ruby 処理系ではネイティブメソッド実行中に Ruby スレッド切り替えを起こさないことを保障していた．つまり，既存のネイティブメソッドは逐次実行を前提としているため，並列実行を行うために必要な排他制御などを含まず，スレッドセーフではない．開発する処理系でこれらのプログラム資産を利用できないのは大きな問題である．

そこで，VM 中の全 Ruby スレッドが共有する単一のロック（ジャイアントロック，以降 GL）を用いて，複数の Ruby スレッドが同時にネイティブメソッドを実行することのないように排他制御を行なう．GL を用いることで旧 Ruby 処理系のために開発されてきたネイティブメソッドをそのまま利用することができる．また，並列度を向上させるために，スレッドセーフに書き換えたネイティブメソッドを GL なしで実行可能

☆ ユーザレベルスレッドをグリーンスレッドとも呼ぶこともある．

にした．スレッドセーフ化を進めることで，段階的に並列度を向上することができる．

本論文の主眼は新規手法の提案ではなく，従来の並列処理をサポートする言語処理系の研究開発とは違い，逐次実行を前提として開発されたネイティブメソッドなど，過去のプログラム資産を利用可能としたまま並列実行による性能向上を実現するという点に着目し，実際に開発を行った上での得られた知見を述べる．

以下，2 章で旧 Ruby 処理系における Ruby スレッド処理機構問題点を述べた後，並列実行可能な Ruby スレッドの実現手法について検討する．3 章でネイティブスレッドに対応する Ruby スレッド処理機構の実現手法を述べる．4 章で並列実行するために必要な，特に同期や排他制御について述べる．5 章で実装した処理系の評価を行い，6 章で関連研究を述べる．7 章でまとめ，今後の課題を示す．

2. Ruby スレッドの並列実行手法の検討

本章では Ruby スレッド処理機構について，旧 Ruby 処理系での実現方式を述べ，その問題点を述べる．そして，その問題点を解決するネイティブスレッドを利用した並列実行可能な Ruby スレッド処理機構の実現方式を検討する．

2.1 旧 Ruby 処理系用スレッド処理機構と問題点

本節では旧 Ruby 処理系での Ruby スレッド処理機構の実装方式とその問題点について述べる．

2.1.1 旧 Ruby 処理系でのスレッド実現手法

旧 Ruby 処理系での Ruby スレッド処理機構は，すべてユーザレベルで実現されており，OS やシステムソフトウェアが提供するネイティブスレッド処理機構は一切利用しない．これは主に移植性を高めるためであり，たとえばネイティブスレッド処理機構を用意していないシステム上でも Ruby を利用すればマルチスレッド並行プログラミングが可能になる．

Ruby スレッドの生成は処理系の Ruby スレッドスケジューラへ新しいスケジューリング対象の登録という形で行われる．これはメモリアクセスのみで行われるため，生成などの操作に OS へのシステムコールを介すことが多いネイティブスレッドよりも高速である．

Ruby スレッドの切り換えはアラームシグナル（SIGVTALRM）などを利用してプリエンプティブに行われる．一定間隔で起動するアラームシグナルハンドラはインタプリタに共通の割り込みフラグをセットする．処理系は定期的に割り込みフラグをポーリングし，Ruby スレッド切り換えのタイミングを検知することができる．ネイティブメソッドの実行中にはポーリングを行わないので切り換えは起こらない．

スレッドコンテキストの切り換えは次のような退避・復帰処理により行われている．まず，setjmp によって取得する実行コンテキスト，およびマシンスタックをヒープに複製して退避する．復帰時はヒープ上に退避していたマシンスタックをスタック領域へ書き戻し，longjmp によって実行コンテキストを回復する．マシンスタックのコピーを行わなくてもマシンスタックの切り換えは可能であるが[19),20)]，そのためにはシステムや CPU に依存したスレッド生成，コンテキスト切り換え処理が必要となり，移植性が低下する．

Ruby スレッドスケジューラは，ある Ruby スレッドの入出力待ちによるブロックによって全スレッドがブロックすることを防ぐため，select システムコールによってポーリングする[26)]．具体的には，スレッドスケジューラが起動すると存在するすべての Ruby スレッドをチェックし，入出力待ちであるスレッドと，入出力待ちをしているファイル記述子を集め，その集合に対して timeout を 0 秒として select システムコールによるポーリングを行う．もし，あるファイル記述子に対してブロックせずに入出力が可能であることがわかれば，そのファイル記述子を待っていた Ruby スレッドを実行可能にする．

また，一時停止中の Ruby スレッドについても Ruby スレッドスケジューラが実行を開始するかどうかを管理している．

2.1.2 解決すべき問題点

旧 Ruby 処理系による Ruby スレッド処理機構が解決するべき問題点は，性能上の問題点と機能的問題点に分けられる．

性能的な問題点としては，すべての Ruby スレッドが一つのネイティブスレッド上で切り換えられながら実行しているため，並列実行による性能向上を行うことができないという点が挙げられる．また，コンテキスト切り換えのたびにマシンスタックのコピーを行うのは性能上問題である．そして，スレッドスケジューリングを行うたびにすべての Ruby スレッドをチェックするのは，Ruby スレッド数が増えれば増えるほど実行時間が長くなり問題である．

機能的な問題点は，Ruby スレッドの真の並行実行に対応できないという点である．これは，ネイティブメソッド実行中にブロックしてしまうような処理や計算時間のかかる処理をある Ruby スレッドが実行した場合，Ruby スレッドスケジューラを起動できなくなるため，全 Ruby スレッドがブロックしてしまうからである．入出力待ちによるブロックは Ruby スレッド

スケジューラにおいてポーリングすることで全 Ruby スレッドのブロックを防いでいる．しかし，その他の要因によるブロックが発生した場合は正しく並行実行することが出来ない．

まとめると，解決すべき問題点は次のようになる．

(1) 並列計算機による性能向上が困難
(2) 高いオーバヘッドのコンテキスト切り換え
(3) スケジューラのスケーラビリティの低さ
(4) ブロックするようなネイティブメソッドの記述が不可能

2.2 ネイティブスレッドを利用した Ruby スレッド実現手法の検討

前節で述べた Ruby スレッド処理機構の問題点を解決するため，OS などシステムが提供するネイティブスレッド処理機構を利用する手法を検討する．

2.2.1 ネイティブスレッドの利用に関する議論

前節で述べた問題点を解決するために，特に並列計算機の利点を生かすためにはネイティブスレッドを積極的に利用するのが自然である．並列計算機環境で提供されるネイティブスレッドはそれぞれ並列に実行可能であることが多いからである．

しかし，ネイティブスレッド処理機構を利用することを前提とした場合，それをサポートしていないシステムでの Ruby スレッドの利用が出来なくなるという移植性の問題が考えられる．これに関しては，最近のほとんどのシステムではネイティブスレッド処理機構を提供しているため，大きな問題はないと判断した．たとえば，UNIX 系 OS での Pthread，Microsoft の Windows での独自のスレッドシステムがある．また，現在広く利用されているネイティブスレッド処理系は事実上，上記の二つなので，移植性の点で大きな問題ではないと判断した．

ネイティブスレッドを採用する欠点は，他にもユーザレベルでの独自スレッド制御に比べて，(a) スレッド制御コストが増大する可能性，および (b) 生成可能なスレッド数が一般的に少ない，という点がある．

(a) に関してはネイティブスレッドの生成や終了などの制御にシステムコールを発行する場合が多いため，一般的に制御コストは高くなる．しかし，ネイティブスレッドを利用すれば環境に依存した高速な Ruby スレッド切り換えを実現できる．Ruby スレッドの生成や終了は，プログラムの最初と最後にまとめて行うようにすることができるが，スレッド切り換えは常に発生する．そのため，ユーザレベルスレッドと比べ，スレッド切り換えの高速なネイティブスレッドのほうが性能的に有利であると判断した．

(b) の生成可能スレッド数は，ユーザレベルで独自にスレッドを実装する場合，生成可能 Ruby スレッド数の制限はメモリサイズのみとなる．しかし，ネイティブスレッドの生成可能スレッド数にはシステム固有の制限があるため，より少数のスレッドしか生成できない．実際に実験したところ，数百から数千スレッドの生成に制限された．しかし，ネットワークプログラムのような Ruby スレッドを利用するアプリケーションにおいて，数千を超える Ruby スレッドを利用することは稀であるため，問題はないと判断した．

2.2.2 Ruby スレッドとネイティブスレッドのマッピング

Ruby スレッドとネイティブスレッドをどのように写像するか，という点では，Ruby スレッド一つにつきネイティブスレッドを一つ用意するのが直観的であり，簡単である（1 対 1 モデル）．コンテキスト切り換えやスレッドスケジューラは，ネイティブスレッド処理機構が提供する効率の良いものを利用することができる．また，あるネイティブスレッドがブロックしても，他のネイティブスレッドは実行を続けることができる．並列実行に対応しているネイティブスレッド処理機構であれば，Ruby スレッドの並列実行による性能向上を得ることができる．つまり，1 対 1 モデルは 2.1.2 で述べた問題点を全て解決する．

このモデル以外にも，M 個の Ruby スレッドにつき，N 個のネイティブスレッドを用意して（$M > N$），M 個の Ruby スレッドを適切なネイティブスレッドに Ruby 処理系側でスケジューリングする（M 対 N モデル）という方式が考えられる．この方式では，スレッド制御をユーザレベルで軽量に行うことができ，並列計算機による性能向上も可能である．しかし，旧 Ruby 処理系と同様に Ruby スレッドをネイティブスレッドに割り当てる処理を独自に実装しなければならず，移植性の問題が発生する．結局，このモデルでは 2.1.2 で述べた問題のうち (2)〜(4) が解決できない．

以上の検討から，1 対 1 モデルを採用することとする．

ただし，Ruby スレッドとネイティブスレッドの対応が 1 対 1 モデルでも，ネイティブスレッド処理機構が Scheduler Activations[2] などの技術を用いた M 対 N モデルのスレッドライブラリであった場合，ユーザレベルでネイティブスレッドの制御を行うこととなり性能が改善する．つまり，ネイティブスレッド処理機構の性能を向上することが Ruby スレッド処理機構の性能向上に直結する．ネイティブスレッド処理機構の性能改善は現在も継続して行われている[14),22)] ため，

今後の技術向上を期待するのは十分妥当である．

2.2.3 同期粒度の検討

ネイティブスレッドを並行・並列に実行するためには，共有資源に対する一貫性保障のための同期や排他制御が必要となる．並列度をあげ，並列計算機上による性能向上を図るためには，これらが必要な箇所を特定し，出来る限り短い同期区間とするべきである．

しかし，すでに存在する膨大なネイティブメソッドの実装は逐次実行を前提としているため，並列実行のために必要な排他制御を行っていない．排他制御を行わないまま Ruby スレッドを並列実行すると，すでに他の Ruby スレッドによって解放したメモリ領域へアクセスしてメモリ保護例外を発生させるなどの，Ruby プロセスを異常終了させるような処理を行う可能性がある．本論文ではこのような異常終了を引き起こす可能性がある処理を危険な処理とし，そうでない処理を安全な処理と定義する．

排他制御は，細粒度ロック（以降 FL）かジャイアントロック（GL）を利用する方法が考えられる．FL は排他制御を必要とする共有資源ごとに用意するロックである．GL は全 Ruby スレッドが共有する単一のロックであり，複数の共有資源をまとめて排他制御する．GL ではなく FL を利用するほうが排他制御の対象が限定的であるために並列度は向上するが，対象ごとにロックを用意する必要があり，実装は困難になる．

ネイティブスレッドを利用するにあたって考えられる，処理系による同期粒度の選択肢は (a) 処理系が FL により排他制御，(b) 処理系が GL により排他制御，(c) 処理系が GL により常に並列実行を抑止，(d) 処理系側では排他制御などを行わずにユーザプログラマが Ruby プログラム上で適切な排他制御を行うことを必須とする，などが考えられる．これらの方法をまとめたものを**図 2** に示す．また，参考までに，旧 Ruby 処理系での Ruby スレッド処理機構を (e) として示す．

図 2 の水平方向の矢印は右方向を正とする時間軸を表し，点線は Ruby スレッドがブロックしていることを表す．RT は Ruby スレッド，NT はネイティブスレッドを示す．NT1, 2 はそれぞれ CPU1, 2 上で実行しているとする．以下，各方式について説明する．

(a) は各共有資源に対して細粒度ロックを用いて処理系による排他制御を行う方式である．図 2(a) では，共有資源 Resource1 と Resource2 に対して RT1，RT2 がそれぞれ並列にアクセスしているが，それぞれ別の細粒度ロック FL1，FL2 を用いて排他制御を行っているため，RT1 と RT2 は並列に実行していることを示している．この方式は性能上有利であるが，既存の

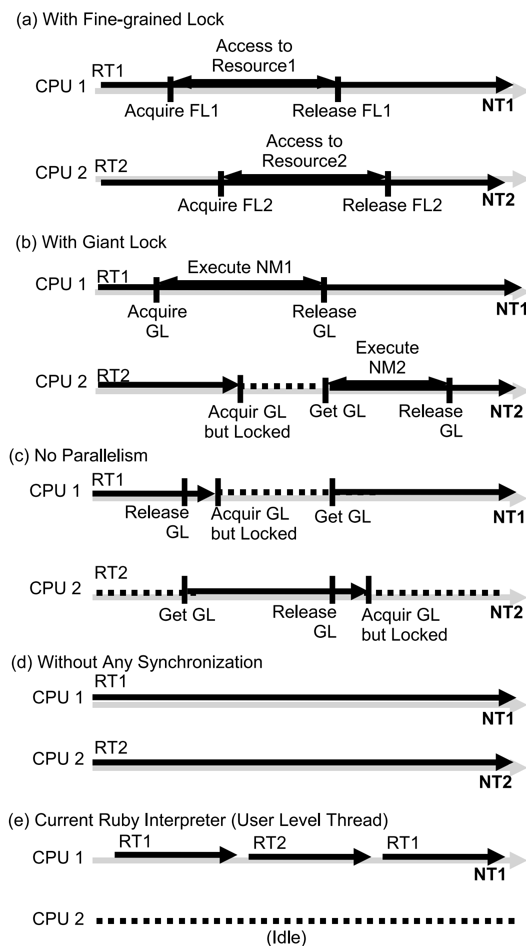

図 2 同期粒度の方式検討
Fig. 2 Synchronization Design

ネイティブメソッドのソースコードを精査し，適切な細粒度ロックの挿入が必要である．この作業は膨大であり，作業も一般的に困難であるため，既存のソフトウェアの再利用という点から，この方式を採用するのは現実的ではない．

(b) は，危険な処理を行う可能性のあるネイティブメソッドを，処理系で GL を獲得してから実行する方式である．図 2(b) は，RT1 がネイティブメソッド NM1 を実行中に RT2 がネイティブメソッド NM2 を実行しようとするが，RT1 が GL を獲得しているため，RT2 が RT1 の GL 解放まで待機してから MN2 を実行することを示している．この方式は，(a) よりも処理系の実装が容易であり，ネイティブメソッド以外は並列実行可能であるという利点があるが，GL による並列度の抑制，GL 制御のオーバヘッドが問題である．

(c) は GL を実行権として扱い，GL を獲得している Ruby スレッドのみしか実行しないよう，処理系が制限する方式である☆．図 2(c) は，GL を保持する Ruby スレッドのみが実行していることを示している．この方式では同時に単一の Ruby スレッドしか実行されないため，排他制御などが不要であり，旧 Ruby 処理系用に開発された多くのプログラム資源が利用可能であるという利点がある．しかし，並列計算機上で並列実行できないという従来の Ruby スレッド処理機構と同じ問題を有する．

(d) は排他制御などを処理系側では行わず，必要な排他制御は Ruby スクリプト記述者が行うとする方式である．この方式は実装が容易であり，性能の点でも最も有利である．なぜなら，(a) ではプログラム上排他制御が不要な共有資源でも必ず排他制御を行うが，(d) では排他制御を行うかどうかは Ruby スクリプト作成者が選択することができるためである．しかし，作成者が排他制御についてすべて責任を負うため，誤って危険な処理を行う可能性がある☆☆．

プログラミング言語 Ruby，および処理系の設計思想として，高速な実行よりも，ユーザに対してなるべく容易な手法を提供するという方針がある．ここでいうユーザとは，Ruby スクリプト作成者はもとより，ネイティブメソッド記述者も含む．方式 (d) は前者，方式 (a) は後者に対して大きな負担を強いることになる．

そこで，方式 (b) を取り，スレッドセーフであると明示しないネイティブメソッドの実行は GL により排他制御することにした．これより，ネイティブメソッドの開発において，並列実行のための特別な処理は記述しなくても済む．つまり，すでに開発されてきた多くのネイティブメソッドがそのまま利用できる．そして，性能に影響する処理を，細粒度ロックを用いた排他制御の追加などを行いスレッドセーフな実装へ書き換えるていくことで，段階的に方式 (a) へ近づけ，並列度を向上させていくことを可能にする．

問題となる GL 獲得のためのオーバヘッドはロック獲得の機会を減少することによって回避する．

2.2.4 検討のまとめ

本章の最後に，本節で述べた検討の結果をまとめる．

開発する Ruby スレッド処理機構はネイティブスレッド処理機構を用いて Ruby スレッドを並列実行可能にする．Ruby スレッド一つにネイティブスレッド一つを割り付ける 1 対 1 モデルとする．スレッドセーフではないネイティブメソッドを実行する際には GL を用いて排他制御し，既存のネイティブメソッドの実装を用いた Ruby プログラムを安全に実行できるようにする．ただし，スレッドセーフに書き換えたネイティブメソッドは並列に実行させる．これにより段階的な並列化による性能向上を可能にする．

☆ この方式を採用しているスクリプト言語 Python[7] の処理系では，ここでいう実行権をジャイアントインタプリタロックと言うが，(b) における GL とは意味が違う．

☆☆ (a)〜(c) についても，トランザクション処理などにおいては Ruby スクリプト記述者による排他制御の挿入が必要である．しかし，これを怠っても危険な処理を行う可能性は無い．

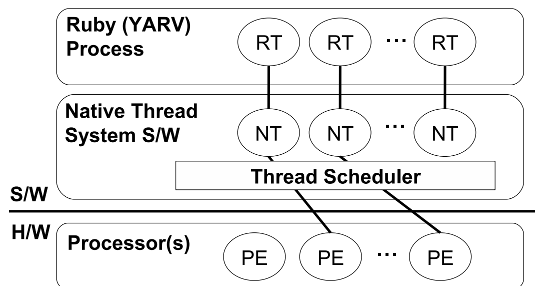

図 3 Ruby スレッドシステムの全体像
Fig. 3 Ruby Thread System Overview

3. ネイティブスレッドを利用した Ruby スレッドの実現

本章では，Ruby スレッドにネイティブスレッドを割り当てる方法について，Ruby スレッド特有の機能を実現する方法とあわせて具体的に述べる．

3.1 Ruby スレッドシステムの全体像

前章の検討の結果，YARV では Ruby スレッドとネイティブスレッドを 1 対 1 対応させるモデルを採用することにした．**図 3** に YARV の Ruby スレッドシステム，ネイティブスレッド処理機構，並列計算機を含めた全体像を示す．

図 3 の RT は Ruby スレッド，NT はネイティブスレッドに対応する．PE は Processor Element の略で，計算機環境の並列実行単位を表し，PE の数だけ並列度があることを示す．Ruby スレッドに対応するネイティブスレッドはネイティブスレッドスケジューラによって，並列実行単位に割り付けられ，並列実行する．

3.2 Ruby スレッド管理データ

YARV では Ruby スレッドを管理するために，スレッド管理データを各 Ruby スレッドごとに保持している．スレッド管理データについて，簡略化したものを**図 4** に C 言語風の擬似コードで示す．

`thread_id` は対応するネイティブスレッドを示す識別子である．Pthread 環境では，型 `thread_id_t` は `pthread_t` の意味である．`status` は Ruby スレッド

```
struct thread_data {
  thread_id_t thread_id;
  int status;
  struct thread_data *wait_next_id;
  lock_t lock; void (*unblock_func)();
  cond_t sleep_cond;
  // その他，VM 実行コンテキスト
  // （スタック，仮想レジスタなど）
};
```

図 4 スレッド管理データ
Fig. 4 Thread Management Data

の実行状況であり，実行中やブロック中，終了後などの情報を保持する．wait_next_id は次節で述べるスレッドの合流にて利用される．lock，unblock_func は 3.5 節で述べるブロック解除関数，sleep_cond は 3.7 節で述べる一時停止に利用される．

その他にスレッド管理データは VM 実行コンテキストを保持している．これは，その Ruby スレッドを実行するために必要なスタックやプログラムカウンタ，スタックポインタなどを意味する．仮想マシンはこの VM 実行コンテキストに随時アクセスしながら Ruby プログラムを実行する．

3.3 Ruby スレッドの制御手法

図 1 で述べた生成，合流，排他制御を実現する Ruby スレッド制御について，ネイティブスレッド処理機構を用いる方法を述べる．

Ruby スレッド制御は仮想マシンの命令ではなく，対応するネイティブメソッドにより実現している．以下，Ruby スレッド制御について，対応するネイティブメソッドとその実現手法を述べる．

生成 (Thread.new) Ruby スレッドの生成時に，対応するネイティブスレッドの生成を行う．Pthread では pthread_create，Windows スレッドでは _beginthreadex API を利用した．生成されたネイティブスレッドは Ruby スレッドとして実行を開始する．

終了 (Thread#exit) Ruby スレッドの実行が終了すると，後処理を終わらせてネイティブスレッド自体も終了するように実装した．ネイティブスレッドが終了・消滅しても，Ruby レベルでアクセスできるスレッドオブジェクト自体はオブジェクトへの参照がある限り生き続ける．

合流 (Thread#join) スレッドの合流はネイティブスレッド処理機構が提供する手段（たとえば，Pthread の pthread_join）は用いず，スレッド管理データに待ちスレッドとして登録（3.2 節の wait_next_tid を利用）するようにして独自に実装した．これは，システムの資源であるネイティブスレッドを，Ruby スレッドの実行終了後，可能な限り早く解放するためである．すでにネイティブスレッドが終了している場合，すぐに合流処理は成功する．

排他制御 (Mutex#synchronize) 排他制御は，ネイティブスレッドが提供する排他制御機能を利用して実装した．具体的には，Pthread 環境では pthread_mutex_lock / unlock，Windows スレッドでは Enter / LeaveCriticalSection を利用した．Windows スレッドにおいて，Mutex オブジェクトを利用しなかったのは，CriticalSection のほうが軽量であること，そして Ruby のスレッド制御では Mutex オブジェクトが提供するプロセス間での同期機能などが必要なかったためである．

```
th = Thread.new do
  ... # (A)
end
# (A) の処理を実行中に例外を発生させる
th.raise SomeException
```

図 5 他の Ruby スレッドに対し例外を発生するプログラム
Fig. 5 Raise an Exception to another Ruby Thread

3.4 Ruby スレッドのスケジューリング

Ruby スレッドの切り換え，スケジューリング処理はネイティブスレッド処理機構の該当機能をそのまま利用する．コンテキスト切り換えは，旧 Ruby 処理系が行っていた非効率な方式ではなく，ネイティブスレッド処理機構によるシステムに依存した高速なコンテキスト切り換えを行うことが期待できる．

Ruby スレッドのスケジューリングの公平性は，ネイティブスレッド処理機構の行うスレッドスケジューリングに依存する．Ruby スレッドに設定された優先度は，ネイティブスレッド処理機構が提供する優先度設定に適切にマッピングする．

3.5 Ruby スレッドへの割り込み

Ruby スレッドの特徴として，任意のタイミングで特定の Ruby スレッドに対して割り込みをかけ，特定の処理を行わせるという機能がある．たとえば，任意の Ruby スレッドに対して，実行中の処理に割り込み強制的に例外を引き起こすことができる（**図 5**）☆．

この割り込み機能は，ポーリングポイントを VM 内に適切に設けることで実現する．割り込みを行う際には，対象の Ruby スレッドに対して割り込み要因を記述してから割り込みフラグをセットする．対象 Ruby

☆ Ruby の標準添付ライブラリの一つである timeout は強制的に例外を引き起こす機能を利用して実装されている．

スレッドはポーリングタイミングで割り込み処理が必要であることを検知することができる．このポーリング間隔が割り込みへの反応速度を決定するが，YARVではVMのジャンプ命令やメソッド起動・終了時にポーリングを行うようにしている．Rubyプログラムでは特にメソッドの起動が頻繁に行われるので，反応時間は十分短い．

ここで問題になるのが，割り込み対象のRubyスレッドがなんらかの理由により一定時間以上ブロックしていた場合である（以降，ブロック状態という）．ブロック状態では割り込みフラグに対するポーリングは行わないため，割り込みフラグのみでは即座に割り込み処理を起こすことが出来ない．

このような問題は，たとえば，入出力待ちのためにブロック状態となったRubyスレッドの処理を，他のRubyスレッドからタイムアウトのために中断したいという場合に発生する．ネットワークプログラミングにおいて，タイムアウトのために例外を発生させ，入出力待ちを中断させるというのは，Rubyではよく行われるイディオムである．

この問題に対処するために，ブロック解除関数を利用することにした．ブロック解除関数とは，ブロックする要因に応じて，そのブロック状態をキャンセルするための処理を行う関数である．あるRubyスレッドRTがブロック状態になる可能性のある処理を実行する前に，ブロック解除関数のポインタをRTのスレッド管理データ（3.2節で述べた`unblock_func`エントリ）に設定する．RTはブロック状態から抜けるときにブロック解除関数の登録を削除する．ブロック状態のRTへ割り込みを行うには，RTに登録されたブロック解除関数を実行する．ブロック状態を脱したRTは割り込みフラグをチェックすることで割り込みを検知する．ブロック解除関数の設定と実行は排他的に行う必要があるが，並行処理の性質上，正当性の検証が容易ではないため，現在も検討中である．

例として，selectシステムコールを実行してブロック状態となったRubyスレッドRTをどのように中断させるか説明する．Pthread環境においてselectシステムコールの実行を中断させるには，`pthread_kill`関数で中断させたいネイティブスレッドに対してシグナルを送るという方式がある．そこで，ネイティブスレッドへシグナルを送るというブロック解除関数`UBF_select`を作成する．RTがselectシステムコールを実行する前に，RTに`UBF_select`を登録する．割り込みを行う場合，RTに登録された`UBF_select`を，RTを引数として実行し，selectシステムコールを中断させる．これらの処理をC言語風の擬似コードにまとめたものを**図6**に示す．なお，図では排他制御に関する処理は省略してある．

```
// select システムコールのためのブロック解除関数
UBF_select(thread_data *th){
  while (th はブロック状態？) {
    pthread_kill(th->thread_id, SIGVTALRM);
  }
}
// select システムコールを実行するネイティブメソッド
ruby_select(thread_data *th){
  set_unblock_function(th, UBF_select);
  // select システムコールを発行・ブロック状態へ
  select(...);
  // select システムコール・ブロック状態から抜ける
  clear_unblock_function(th);
  CHECK_INTERRUPT(th);
}
// 割り込みをかける処理
interrupt(struct thread_data *th, intr_t *intr){
  SET_INTERRUPT(th, intr); // 割り込み要因の設定
  if (th->unblock_function)
    (th->unblock_function)(th);
}
```

図6 ブロック解除関数とその利用例
Fig. 6 Examples of Unblock Function and That Usage

本節冒頭で述べた，他Rubyスレッドへの例外発生は，YARVによって次のように行われる．まず例外を発生させるという情報を対象Rubyスレッドへ登録し，対象Rubyスレッドのブロック解除関数が登録されていればそれを実行する．対象Rubyスレッドは例外が配送されたことを検知して例外処理を発生する．

3.6 シグナルの扱い

シグナルはUNIXでのプロセス間通信などに用いられるが，Rubyレベルでもシグナルはサポートしている．つまり，Rubyプロセスが受信したシグナルに応じて実行するべき処理をRubyプログラムで記述することができる．シグナルに関してはシステムによって扱いが異なるため，ここではPthread環境についてのみ説明する．

Pthread環境ではネイティブスレッドごとにシグナルマスクを持つ．プロセスに送られたシグナルは，そのシグナルがマスクされていないネイティブスレッドのうちどれか一つに配送される．任意のRubyスレッドがシグナルを受信可能とすると，任意のタイミングでシグナルを受信することを考慮しなければならず，実装が複雑になる．また，Pthreadの条件変数による一時停止などは，シグナルハンドラの実行によって解除できないため，シグナルに対応する処理を正しく実行することが出来ない可能性がある．

そこで，Rubyスレッドに割り当てるネイティブス

```
// 一時停止を解除するブロック解除関数
UBF_sleep(thread_data *th){
  pthread_cond_signal(th->sleep_cond, ...);
}
// 一時停止するネイティブメソッド
ruby_sleep(thread_data *th, timeval *t){
  set_unblock_function(th, UBF_sleep);
  if (t == 0) // 割り込みがあるまでブロック
    pthread_cond_wait(th->sleep_cond, ...);
  else        // 指定時間までブロック
    pthread_cond_timedwait(th->sleep_cond, ...);
  clear_unblock_function(th);
  CHECK_INTERRUPT_FLAG(th);
}
```

図 7 Pthread 環境での一時停止処理
Fig. 7 An program of sleep with Pthread

レッド以外に，管理スレッドというネイティブスレッドを用意した．管理スレッド以外のネイティブスレッドのシグナルマスクを設定し，管理スレッドにのみシグナルが配送されるようにした．管理スレッドがシグナルを受信すると，即座にメインスレッド（Ruby プログラム開始と同時に生成される Ruby スレッド）へ前節で述べた割り込みをかけ，受信したシグナルに対応する処理を実行させる．

3.7 Ruby スレッドの一時停止

YARV では `sleep` メソッドなどで Ruby スレッドを一時停止させるために，Pthread 環境下では条件変数と `pthread_cond_timedwait`，もしくは `pthread_cond_wait` 関数を利用して一時停止を実現した．前者は時間制限のある一時停止（いわゆる sleep），後者は他 Ruby スレッドからの割り込みなどがあるまで中断する場合に利用する．ブロック解除関数は `pthread_cond_signal` 関数を呼び出す．**図 7** に Pthread 環境での一時停止処理を C 言語風擬似コードで示す．

Windows の場合は，`WaitForMultipleObjects` でイベントオブジェクトを待ち合わせることで実現した．ブロック解除関数は `SetEvent` 関数でイベントオブジェクトをシグナル状態にすることで実現した．

`Thread#join` での Ruby スレッドの合流にもこの一時停止の機構を利用した．

4. Ruby スレッドの並列実行

本章ではネイティブスレッドに割り当てた Ruby スレッドを並列実行するために必要な点について述べる．

4.1 スレッドローカルストレージの利用

Ruby スレッドを並列実行するためには，VM コンテキストを参照するために，各 Ruby スレッド管理データへ同時にアクセスする必要がある．

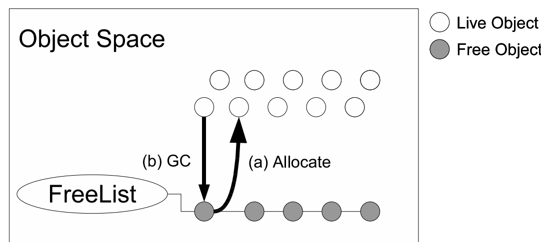

図 8 旧 Ruby 処理系でのメモリ管理
Fig. 8 Memory Management on Current Ruby Interpreter

これを実現するために，ネイティブスレッド処理機構がサポートするスレッドローカルストレージ（以降，TLS）を利用した．TLS はネイティブスレッドごとにそれぞれ別々にもつ空間である．各 Ruby スレッドはそれぞれのスレッド管理データへのポインタを，対応するネイティブスレッドの TLS に格納する．

TLS の利用方法はネイティブスレッド処理機構によって様々であるが，たとえば Pthread の規格のみに沿うのであれば `pthread_getspecific` 関数などが利用できる．また，OS と C コンパイラの組み合わせによっては通常のグローバル変数定義に `thread`，もしくは`__thread` 指示子を付加することで，その変数をスレッドローカル変数として利用できる．

4.2 並列実行に対応したメモリ管理

Ruby はガーベージコレクション（以降，GC）によるメモリ管理を行うが，並列実行を行うために排他制御や同期を行わなければならない．そこで，本節では YARV が行う並列実行に対応したメモリ管理について述べる．なお，今回は既存のメモリ管理を拡張する形で実装を行ったため，旧 Ruby 処理系でのメモリ管理とあわせて説明する．

4.2.1 旧 Ruby 処理系のメモリ管理

ここで，説明のために旧 Ruby 処理系のメモリ管理を簡単に紹介する．Ruby でのメモリ管理はオブジェクト単位で行うので，オブジェクトの管理方式について説明する．

図 8 に示すとおり，オブジェクトスペースには生きているオブジェクト（図中白丸）と解放されたオブジェクト（図中灰色の丸）がある．解放されたオブジェクトは FreeList という連結リストによって管理されている．生きているオブジェクト，解放されたオブジェクト，および FreeList はどの Ruby スレッドからも参照可能である．

オブジェクト割り当て時には FreeList の先頭のオブジェクトを新しいオブジェクトとして割り当てる（図中 (a)）．この処理は連結リストの操作として行われる．

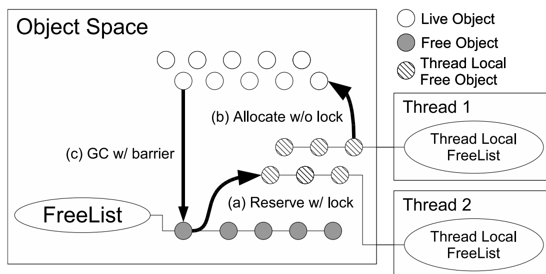

図 9 並列実行に対応したメモリ管理
Fig. 9 Memory Management on Parallel Execution

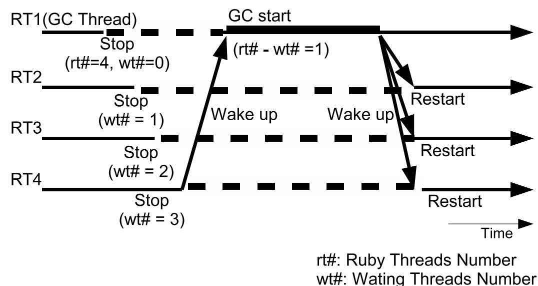

図 10 GC のための同期
Fig. 10 Synchronization for Garbage Collection

GC アルゴリズムは停止型の保守的マークアンドスイープである[25]．これは，ネイティブメソッド実装時にリードバリアやライトバリアが必要ないなど，実装のしやすさを優先するための選択である．

GC はマシンスタックなどを根としてマークする．マークされなかったオブジェクトが回収され FreeList につながれる（図中 (b)）．

旧 Ruby 処理系ではメモリ管理の実行中に他の Ruby スレッドへ切り換わることが無いように実装されているため排他制御は一切不要である．

4.2.2 オブジェクトアロケーション

Ruby スレッドが並列実行する場合，FreeList はどの Ruby スレッドからも同時にアクセスされる可能性があるため，FreeList に対するリスト操作には排他制御が必要である．しかし，Ruby プログラムにおいて頻発するオブジェクトアロケーションのたびに排他制御を行うのは性能上問題である．そこで，スレッドローカルな FreeList（以降，TLFL）を用いることでこの問題を解決した．この方式を**図 9** に示す．

まず，Ruby スレッドは FreeList から，解放されたオブジェクトを N 個（現在の実装では $N = 4096$）確保し，TLFL へ連結する（図 9 の (a)）．このとき，FreeList のリスト操作は排他制御を伴う．

オブジェクトの割り当て時には，TLFL の先頭から一つオブジェクトを取り出して割り当て処理を行う．TLFL は他の Ruby スレッドからはアクセスされないため排他制御は不要である．もし TLFL が空だった場合，再度 FreeList から一定数確保する．

この工夫により，オブジェクト確保の際に排他制御が必要になる機会を $1/N$ 回に削減している．

4.2.3 ガーベージコレクション

YARV では旧 Ruby 処理系と同様の理由で停止型の保守的マークアンドスイープ GC を採用している．

停止型 GC を実現するため，GC 時には全 Ruby スレッドを停止する．Ruby スレッド数 4 で，GC 実行 Ruby スレッド RT1 が GC を行う様子を**図 10** に示す．RT1 は他の Ruby スレッド全てが一時停止するまで，つまり (全実行スレッド数 $rt\#$ − 待機スレッド数 $wt\#$) が 1 になるまで待機する．RT2〜RT4 は同期のために wt#を 1 ずつ増加させてから一時停止する．RT4 が一時停止する際，$rt\# - wt\# = 1$ となるため RT1 の実行を再開させる．RT1 は GC を実行（図 9(c)）し，GC 終了後に GC のために一時停止していた全 Ruby スレッドの実行を再開する．

Ruby スレッドは，前章で述べた割り込みフラグポーリングポイントにおいて GC 処理が発生中であるかをチェックし，同期を行う．同期は Pthread 環境での条件変数，Windows 環境でのイベントハンドルを利用して実現した．

ブロック状態の Ruby スレッドは GC のための待ちを行うことが出来ないので，ブロック状態になる前に待機スレッド数を増加させておく．そのとき，マークに必要となるマシンスタックなどの情報も同時に保存しておく．GC 実行中にブロックが解除されたときには GC が終了するまで待機する．GC のための同期中に Ruby スレッドが終了した場合は，全実行スレッド数を 1 減らして GC スレッドに通知する．

4.3 排他制御の導入

本節では並列実行にあたって必要となった排他制御の導入について，特にハッシュ表の排他制御，およびジャイアントロックの導入について述べる．

4.3.1 ハッシュ表の排他制御

旧 Ruby 処理系，および YARV が利用しているハッシュ表の実装では，参照と設定処理はアトミックに行わなければならない．そこで，ハッシュ表ごとに細粒度ロックを設けて，ハッシュ表に対する参照・設定時には細粒度ロックを用いて排他制御を行うようにした．

YARV において，グローバル変数，インスタンス変数，定数，メソッドテーブルなどの並列に利用される可能性がある機能はハッシュ表を用いて実現している．

ハッシュ表へのアクセスをスレッドセーフにしたことにより，これらの機能もスレッドセーフとなった．

4.3.2 ジャイアントロックの導入

既存のネイティブメソッドは逐次実行を前提としておりスレッドセーフではないので，複数のネイティブメソッドを同時並列に実行すると Ruby プロセスの異常終了を引き起こすような危険な処理を行う可能性がある．そこで，2.2.3 で検討したとおり，そのような恐れのあるネイティブメソッドは，VM につき一つ用意するジャイアントロック（GL）を獲得してから実行するようにした．

GL 獲得のタイミングはスレッドセーフと明示的に示されていないネイティブメソッドの起動時に行う．なお，明示する方法については次項で述べる．獲得した GL は，ネイティブメソッド終了時に解放する．この方式により，すでに多く存在する Ruby 用拡張ライブラリを安全に利用することができる．

ただし，ネイティブメソッド実行中に Ruby スクリプトで定義されたメソッドを呼び出すことができるので，その際は GL を解除してメソッドを実行する．メソッドが処理が戻った時，GL を再獲得する．

また，YARV では例外処理のために `longjmp` 関数によって大域ジャンプを行う場合がある[8]．このとき GL を獲得していない状態で大域ジャンプしたが，ジャンプ先を記録した時点（`setjmp` した時点）では GL を獲得していた場合，GL 獲得状態に不整合が生じる．そのため，`setjmp` 関数によって実行コンテキストを保存する際には同時に GL 獲得状態も保存しておき，大域ジャンプによって GL 獲得状態の不整合が起こることを防ぐ．

4.3.3 スレッドセーフな処理を宣言するための API

ネイティブメソッドをスレッドセーフであると明示するために `rb_define_method_ts` 関数（ts: Thread Safe）を用意した．この API で定義されたネイティブメソッドは，呼び出し時に GL 獲得操作を行わない．

スレッドセーフなネイティブメソッドを段階的に増やしていくことで，YARV の並列度を順次向上することができる．

4.4 排他制御オーバヘッドを削減するための工夫

排他制御を行うと，ロック獲得，解除のためのオーバヘッド，および競合状態におけるネイティブスレッドの待ち状態への遷移にかかるコストが問題になる．また，現在の YARV の実装ではスレッドセーフなネイティブメソッドが少ないため，GL 獲得操作は比較的多く行われる．したがって，排他制御のオーバヘッド削減の工夫は重要である．

```
/* (A) 参照時に排他制御を用いる手法 */
cache_entry method_cache[CACHE_SIZE];
method_search_with_cache(class, id){
  LOCK(method_cache_lock); // ロック獲得
  cache_entry *e = &method_cache[HASH(class, id)];
  if (!(e->class == class && e->id == id)) {
    /* キャッシュミスのため，メソッド探索を行い
       メソッドキャッシュエントリを更新 */
    e->class = class; e->id = id;
    e->body = method_search(class, id);
  }
  UNLOCK(method_cache_lock); // ロック解放
  return e->body;
}
/* (B) 参照時に排他制御が不要な手法 */
cache_entry *method_cache[CACHE_SIZE];
method_search_with_cache(class, id){
  cache_entry *e = method_cache[HASH(class, id)];
  if (!(e->class == class && e->id == id)) {
    /* キャッシュミスのため，メソッド探索を行い
       新しいメソッドキャッシュエントリを作成 */
    method_cache[HASH(class, id)] = e = new_entry(
      class, id, method_search(class, id));
  }
  return e->body;
}
```

図 11 メソッドキャッシュ実現手法
Fig. 11 Method Cache Implementations

そこで，本節では排他制御オーバヘッドを削減するための工夫について述べる．

4.4.1 ロック不要なメソッドキャッシュの参照

YARV ではグローバルメソッドキャッシュ，およびインラインメソッドキャッシュを利用している[23]．キャッシュ表の参照はキャッシュエントリとしてメソッド ID やクラスをキーとし，メソッド本体を値としている．このエントリの更新，および参照はアトミックに行わなければ不整合を生じる．**図 11** の (A) にロックを用いたメソッド検索処理を C 言語風の擬似コードで示す．このプログラムではメソッド探索を行う間，メソッドキャッシュ全体をロックする．しかし，Ruby プログラムではメソッド起動は頻繁に行われるため，メソッドキャッシュ参照のたびに排他制御を行うのは性能上問題である．

そこで，キャッシュミス時にはメソッドキャッシュエントリ自体をアトミックに切り替えることで排他制御を不要とした．この処理を図 11 の (B) に示す．キャッシュミス時にエントリの内容を書き換える際，キャッシュエントリを GC 対象オブジェクトとして新しく生成し，キャッシュテーブルに登録する．キャッシュエントリの参照とキャッシュ表への登録はポインタアクセスであるためアトミックに行うことができる．した

がって参照時に排他制御を行う必要はない．

なお，図 11(B) の処理順序は意味があるので，コンパイラによる最適化等で入れ替わらないように工夫する必要がある他，プロセッサによってはメモリオーダリングに応じた適切なメモリバリア命令の挿入などが必要となるので注意されたい．

キャッシュミス時は新たに GC 対象オブジェクトの割り当てが起こりコストがかかるが，ヒット時は排他制御が必要ない．メソッドキャッシュのヒット率は十分高い[23] ため，採用した手法のほうが有利である．

ただし，プログラムによってはインラインメソッドキャッシュのヒット率が低くなる場合があるため，ミスを繰り返したら今後そのメソッド呼び出しではインラインメソッドキャッシュを利用しないようにした．

4.4.2 1 スレッド実行時のジャイアントロック

Ruby スレッドが一つしかない場合，GL による排他制御を行う必要はないので行わない．

Ruby スレッドが一つであるかどうかの判断は，共有資源である同時実行スレッド数カウンタを参照する必要がある．しかし，そのカウンタが 1 であれば，ほかの Ruby スレッドがその値を変更する可能性はないため，排他制御無しにこの確認を行うことができる．厳密には，タイミングによって Ruby スレッド数を 2 以上と誤判定する可能性があるが，通常の GL 獲得処理を行うだけなので問題ない．

また，単一 Ruby スレッド実行時に新たな Ruby スレッド生成時，GL を獲得してから生成処理を行うことで，GL 獲得状態の一貫性を保つことができる．

4.4.3 スピンロックの利用

GL 獲得時，他の Ruby スレッドによって GL がすでに獲得されていた場合，競合が発生する．事前評価によると，Pthread 環境にてネイティブスレッドを待ち状態へ移行するとき，実行時間がかかることがわかった．そこでスピンロックを積極的に利用するようにした．本項では Pthread 環境での実装例を紹介する．

まず，`pthread_mutex_trylock` 関数を用いて一定回数（現在の実装では 100 回）スピンロックすることにした．各繰り返しでは `sched_yield` 関数によってプロセッサの実行権を手放し，他の Ruby スレッドに制御を移すことで GL を獲得している Ruby スレッドの実行を先に行うようにした．

また，`pthread_mutex_trylock` の実行回数も削減するために GL が解放されているかどうかのチェックも行った．これをまとめ，Pthread 環境での GL 獲得処理を C 言語風の擬似コードで記述したものを**図 12** に示す．なお，GL 解放時には `gl_locked` 変数を 0 クリアする．

```
pthread_mutex_t giant_lock;
int gl_locked;
acquire_giantlock(){
  for (int i=0; i<100; i++){
    if (gl_locked == 0 &&
        pthread_mutex_trylock(&giant_lock) == 0) {
        gl_locked = 1; return;
    } sched_yield();
  } // 競合発生
  pthread_mutex_lock(&giant_lock);
  gl_locked = 1;
}
```

図 12 ジャイアントロック獲得処理（Pthread 環境）
Fig. 12 A Program to Acquire Giant Lock with Pthread

4.4.4 利用 CPU の制限

GL 獲得のための競合が多発し，前述したネイティブスレッドの待ち状態への遷移が頻発すると大きく性能が低下する．これは，複数の Ruby スレッドが頻繁に GL 獲得を行うことで生じる．

このような状況では，競合を起こしている Ruby スレッドがその後も競合を発生すると予測できる．そこで，競合を頻発させる Ruby スレッド群は並列実行しないようにした．具体的には，競合を繰り返す Ruby スレッド群が利用できる CPU を一つに制限することで実現した．現在の実装では，制限する Ruby スレッド群を 1 秒間に GL 競合回数が 3000 回を越える Ruby スレッドの集合としている．この数値は環境や動作させるプログラムによって最適な値が異なるため，YARV 起動時に指定出来るようにした．この機能は `pthread_setaffinity_np()` 関数（NPTL[3]），または `SetThreadAffinityMask()` 関数（Windows）を用いて実装した．ただし，これらの関数をサポートしていないシステムではこの機能は無効となる．

なお，利用プロセッサを制限した後，プログラムの挙動が変化し，競合が起こらないようになっている可能性があるので，この制限は一定の間隔（現在の実装では 1 秒ごと）で解除するようにした．

4.4.5 単一実行権による逐次実行

並列計算機ではない計算機システムや，本質的に並列計算できないプログラムの場合，並列実行による性能向上が得られないため，並列実行は排他制御のオーバヘッドの分だけ無駄である．

そのため，GL を単一実行権として扱い，Ruby スレッド切り換え時に GL を解放，つまり実行権を手放し，他の Ruby スレッドへ遷移するというモード（コンパイルオプション）を設けた．これは，2.2.3 において検討した方式 (c) を実装したものである．

このモードでは並列実行を行うことができないが，

表 1 スレッド制御プリミティブの性能評価
Table 1 Evaluation of Thread Management Primitives

	Ruby	YARV
生成（10 万回生成に要した秒数）	1.82	5.06
合流（10 万回合流に要した秒数）	0.58	2.32
排他制御（100 万回行った秒数）	3.32	0.43

実行時に処理系による排他制御をほぼ行わない．並列実行による性能向上が見込めない場合，このモードが性能的に有利である．

5. 評　　価

本章では，ネイティブスレッドに対応した Ruby スレッドの性能と，並列性の評価について述べる．

評価環境は，Intel Pentium D CPU 3.46GHz プロセッサ（Dual Core），上で動作する Linux 2.6.17-1.2174_FC5 SMP を用いて行った．利用したコンパイラは gcc version 4.1.1 20060525 (Red Hat 4.1.1-1) である．比較対象とする Ruby 処理系は ruby 1.9.0 (2006-04-08) [x86_64-linux] を用いた．Linux 環境下であるため，利用するネイティブスレッド処理機構は Pthread 環境（NPTL）である．YARV の最適化オプションは文献 8) で述べた最適化オプションで融合操作，スタックキャッシング最適化以外を適用した．

評価は複数回実行し，もっとも速いものを計測結果とした．

5.1 スレッド制御プリミティブの評価

本節では Ruby スレッド制御の性能評価を行う．

5.1.1 スレッド生成，合流，同期

Ruby スレッド制御プリミティブの実行時間を**表 1** に示す．

Ruby スレッドの生成はネイティブスレッドの生成コスト（この場合，`pthread_create` のコスト）がそのままかかるので，性能が悪い．また，合流についても，ネイティブスレッドによる同期処理（この場合 `pthread_cond_wait`）が必要になるため，旧 Ruby 処理系に比べて性能は低い．

排他制御は `Mutex` オブジェクトを利用したロックの獲得と解除を繰り返し行ったものである．YARV のほうが性能が高い理由は，旧 Ruby 処理系は Mutex クラスの処理を Ruby レベルで記述しており，YARV では C 言語レベルで組込みクラスとして実装しているためである．

一般的に，ある程度長期間生きる Ruby スレッドを利用するプログラムでは，スレッド生成コストよりも排他制御のコストのほうが問題になるため，YARV のほうが有利であると言える．

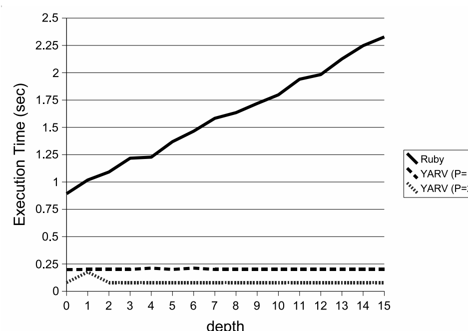

図 13 コンテキストスイッチの実行時間
Fig. 13 Overhead of Thread Context Switching

なお，旧 Ruby 処理系においては，同期，排他制御のプリミティブとして `Thread.critical` の設定がある．これは，セットすると他の Ruby スレッドへのスイッチを禁止するという機能である．しかし，この機能にはいくつか問題があり，並列実行に対応するにはコストが大きすぎるため廃止することが決定している．そのため，本論文では `Thread.critical` による排他制御は評価に含めなかった．

5.1.2 スレッド切り換え

Ruby スレッドを二つ用意し，それぞれスレッド切り換え処理を 10 万回行い，その速度を計測した．旧 Ruby 処理系のスレッド切り換えにはマシンスタックのコピーを伴うため，スタックの深さに処理時間が比例するという問題があった．そこで，**図 13** で示すように，スレッド切り換え時におけるスタックの深さをパラメータとしてスレッド切り換えの処理時間を計測した．評価には旧 Ruby 処理系 (Ruby) と，利用するプロセッサ数が 1 の場合である YARV (P=1) と 2 の場合である YARV (P=2) を用いた．

評価の結果，旧 Ruby 処理系はスレッド切り換えのコストがスタックの深さに比例するが，YARV のコストは一定であり，旧 Ruby 処理系よりも軽量であることがわかった．また，YARV (P=1) と YARV (P=2) を比較した場合，YARV(P=2) が 2 倍以上高速であり，並列実行の効果が出ていることが確認できた．

5.2 マイクロベンチマーク

図 14 にマイクロベンチマークの結果を掲載する．このグラフでは，縦軸を単一 CPU で実行する YARV (P=1) の実行時間を基準とした速度向上比としている．fib はフィボナッチ数を再帰処理で求めるプログラム，tak は tak 関数，concat は文字列の追加の繰り返し，mandelbrot はマンデルブロ集合を求めるプログラムである．それぞれ Ruby スレッドを 4 つ，同

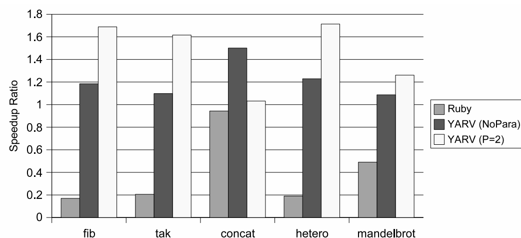

図 14 マイクロベンチマークの結果
Fig. 14 Result of Micro-Benchmarks

じ処理を実行した．ただし，concat は 4 スレッドが同時に一つの文字列に追加するため競合が発生する．hetero は Ruby スレッド二つを fib，Ruby スレッド二つを concat の処理にあて，同時に実行する．

この評価では比較対象として前節で述べた旧 Ruby 処理系 (Ruby)，2 CPU で並列実行する YARV (P=2) に，YARV (NoPara) を加えた．YARV (NoPara) は 4.4.5 で述べた並列実行を行わない代わりに排他制御のオーバヘッドを不要とするモードである．

結果を見ると，fib や tak など，数値計算のような互いに独立に実行できる処理が主なものは並列実行による速度向上を確認できた．

しかし，concat のような文字列処理で GL による排他制御が必要な場合☆，並列計算による性能向上は見ることが出来なかった．また，ロック獲得のオーバヘッドが不要であるという点から，YARV (NoPara) がもっともよい性能を示した．mandelbrot では結果を集める処理で GL が必要な処理があったため，YARV (NoPara) に比べて 1.2 倍程度の性能向上しか得られなかった．hetero の結果から，GL が必要となる処理があっても，他スレッドが並列に実行するため性能向上が見られた．

ここで，GL による排他制御が性能にどのような影響を与えるかを確認するため，YARV (P=2) の実行における各プログラムでの GL の獲得頻度，および競合頻度を調査した．この結果を**表 2** に示す．それぞれ GL 獲得回数，GL 競合回数を YARV (P=2) の実行時間（秒）で割ったものである．

fib，tak は GL 獲得回数が少ないため，GL 制御のオーバヘッドは無いことがわかる．concat は GL 獲得・競合頻度が高いため，4.4.4 で示した利用 CPU 制限機能が働き，ほぼ単一 CPU で実行することとなった．そのため，並列実行による性能向上が得られなかった．hetero では，fib の処理と concat の処理には依存関係がないため，concat 処理での GL 獲得頻度は高くても GL 競合頻度が少なくなり，利用 CPU 制限が起こらなかった．このため，並列実行による性能向上を得ることができた．mandelbrot は GL 獲得回数は比較的少ないが，結果を集める処理の一部に長時間 GL を獲得したまま解放しない場合があったため，競合が多く発生した．これが並列実行における性能向上のボトルネックとなった．

この結果から，GL を必要とする処理が多い現状では，hetero のような，GL を利用する Ruby スレッドのほかに GL 不要の計算を行う Ruby スレッドを作り並列度を向上するのが現実的であると考えられる．

☆ 文字列処理に関しては，文字列ごとに細粒度ロックを用意し，これを用いて排他制御することで GL の利用を不要とすることが出来る．しかし，現在の実装では文字列処理のすべてをスレッドセーフな処理に置き換えていないため，ここでは GL による排他制御を行っている．

表 2 GL 獲得・競合頻度
Table 2 Frequency of GL acquirement/competition

ベンチマーク名	GL 獲得頻度	GL 競合頻度
fib	0.01×10^3	1.88
tak	0.03×10^3	4.36
concat	$2,427.06 \times 10^3$	107.67
hetero	$1,000.25 \times 10^3$	0.67
mandelbrot	183.08×10^3	292.6

(times/sec)

6. 関連研究

すでに多くの並列実行をサポートするプログラミング言語とその処理系が研究，開発されている[27]．しかし，それらの言語の処理系は開発当初より並列化を意識した構造としているため，本論文で議論したような，既存のコードを活用するなどの問題の多くはそもそも存在しない．

プログラミング言語 Java[10] ではマルチスレッド，および並列実行に関する研究が多く行われてきており，特に Java 仮想マシン[17] における同期コストの削減のための研究が行われている[18]．Java で実装する Java 仮想マシンである Jalapeno[1] では，本論文では利用しなかった M 対 N モデルを採用しており，プロセッサの命令を利用して同期，排他制御などを実装し，高い性能を得ている．しかし，その方式では移植性に問題があるため，YARV では採用しなかった．4)，6) などの研究では Java 仮想マシンでのネイティブスレッドの利用について，移植性や性能についての議論しているが，それらの知見は本論文で述べた実装にもある程度あてはまる．

スクリプト言語のマルチスレッド対応については，

Perl[12] では Parrot[11] にマルチスレッド機能を取り込む作業を行っているが，仕様を検討中の段階である．現在の Perl 処理系（Perl 5.8）がもつ ithread[9] は，スレッド間で共有するオブジェクトを明示的に指定せねばならず，Ruby のスレッドと等価なものではない．Python[7] では本論文 4.4.5 で述べた単一実行権を用いて同時に一つの Python スレッドしか動作させないというネイティブスレッドを用いたスレッドの実装を行っているが，並列計算機による性能向上を得ることができない．また，Python のガーベージコレクタはリファレンスカウンタで実装しているため，並列実行可能な実装にするにはオーバヘッドがかかる．Ruby，および YARV では停止型の保守的マークアンドスイープ GC を採用しているのでその問題はない．

7. ま と め

本論文では Ruby 処理系である YARV の Ruby スレッド処理機構をネイティブスレッドを利用して実現し，GC などで適切な同期を行い並列実行を可能にした．排他制御を行わない既存のネイティブメソッドを安全に実行するためにジャイアントロック（GL）を導入した．また，GL による競合の頻発を抑えるためにいくつかの工夫を行った．評価の結果，並列計算機上での並列実行による性能向上を確認できた．しかし，ネイティブスレッドの生成にかかるオーバヘッドによる性能低下もおきた．

今後の課題は多いが，特に標準ライブラリにスレッドセーフなネイティブメソッドが少ないので見直しが必要である．中でも Ruby プログラムで頻出する文字列，および配列操作についてスレッドセーフ化を行い処理並列性を向上させることが必要である．また，現在 GC の処理は逐次実行しているが，並列 GC についても導入を検討したい．その他，Ruby スレッドの生成が高速であるという前提で Ruby スレッドを多く生成するような Ruby プログラムが多くあるので，たとえばスレッドプールを利用するような方式に変更していく必要がある．

謝 辞

YARV 開発にあたり，YARV 開発用メーリングリストに参加されている方々にはいつも有益なアドバイスを頂いております．感謝いたします．

本処理系の開発プロジェクトは，IPA（情報処理推進機構）の公募事業 2004 年度未踏ソフトウェア創造事業「未踏ユース」（PM：筧 捷彦早稲田大学教授），および 2005 年度未踏ソフトウェア創造事業（PM：千葉 滋東京工業大学大学院助教授）に採択され，支援を受けました．

参 考 文 献

1) Alpern, B. et al.: The Jalapeno Virtual Machine, *IBM Systems Journal, Java Performance Issue*, Vol. 39, No. 1 (2000).
2) Anderson, T. E., Bershad, B. N., Lazowska, E. D. and Levy, H. M.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79 (1992).
3) Drepper, U. et al.: The new Native POSIX Thread Library for Linux : NPTL. http://people.redhat.com/drepper/nptl-design.pdf : Draft.
4) Gu, Y., Lee, B. S. and Cai, W.: Evaluation of Java Thread Performance on Two Different Multithhreaded Kernels, *Operating Systems Review*, Vol. 33, No. 1, pp. 34–46 (1999).
5) IEEE: *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1* (1996).
6) Pinilla, R. and Gil, M.: JVM: platform independent vs. performance dependent., *Operating Systems Review*, Vol. 37, No. 2, pp. 44–56 (2003).
7) Python Software Foundation: Python Programming Language. http://www.python.org/.
8) Sasada, K.: YARV: Yet Another Ruby VM. http://www.atdot.net/yarv/.
9) Sugalski, D.: perlthrtut - tutorial on threads in Perl. http://search.cpan.org/ jhi/perl-5.8.1/pod/perlthrtut.pod.
10) Sun Microsystems: Java テクノロジ．http://jp.sun.com/java/.
11) The Perl Foundation: Parrot - parrotcode. http://www.parrotcode.org/.
12) The Perl Foundation: The Perl Directory - perl.org. http://www.perl.org/.
13) Thomas, D., Fowler, C. and Hunt, A.: *Programming Ruby*, The Pragmatic Programmers (2004).
14) von Behren, R., Condit, J., Zhou, F., Necula, G. C. and Brewer, E.: Capriccio: scalable threads for internet services, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM Press, pp. 268–281 (2003).
15) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, 株式会社アスキー (1999).
16) まつもとゆきひろ他: オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/.
17) ティム・リンドホルム, フランク・イェリン: Java

仮想マシン仕様第 2 版, ピアソン・エデュケーション (2001).
18) 河内谷, 古関, 小野寺: スレッド局所性を利用した Java ロックの高速化, 情報処理学会論文誌 (PRO), Vol. 44, No. SIG 15 (PRO19), pp. 13–24 (2003).
19) 多田好克, 寺田実: 移植性･拡張性に優れた C のコルーチンライブラリー実現法, 電子情報通信学会論文誌, Vol. J73D-I, No. 12, pp. 961–970 (1990).
20) 安倍広多, 松浦敏雄, 谷口健一: BSD UNIX 上での移植性に優れた軽量プロセス機構の実現, 情報処理学会論文誌, Vol. 36, No. 2, pp. 296–303 (1995).
21) 松本行弘: Ruby の真実, 情報処理, Vol. 44, No. 5, pp. 515–521 (2003).
22) 笹田, 佐藤, 河原, 加藤, 大和, 中條, 並木: マルチスレッドアーキテクチャにおけるスレッドライブラリの実現と評価, 情報処理学会論文誌：ACS, Vol. 44, No. SIG11(ACS3), pp. 215–225 (2003).
23) 笹田耕一: プログラム言語 Ruby におけるメソッドキャッシング手法の検討, 情報処理学会第 67 回全国大会, Vol. 1, pp. 305–306 (2005).
24) 笹田耕一, 松本行弘, 前田敦司, 並木美太郎: Ruby 用仮想マシン YARV の実装と評価, 情報処理学会論文誌 (PRO), Vol. 47, No. SIG 2(PRO28), pp. 57–73 (2006).
25) 青木峰郎: Ruby ソースコード完全解説, インプレス (2002).
26) 田中哲: Ruby I/O 機構の改善 – stdio considered harmful –, Linux Conference 抄録集, pp. 1–10 (2005).
27) 田浦健次朗: 細粒度マルチスレッディングのための言語処理系技術, コンピュータソフトウェア, Vol. 16, No. 2, 3, pp. 1–19, 9–28 (1999).

(平成 18 年 9 月 12 日受付)
(平成 18 年 12 月 12 日採録)

笹田　耕一（正会員）

2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006 年同大学院工学教育部博士後期課程電子情報工学専攻退学．同年東京大学大学院情報理工学系研究科特任助手．現在同特任助教．システムソフトウェア，とくに並列処理システム，言語処理系に関する研究に興味を持つ．

松本　行弘（正会員）

1990 年筑波大学第三学群情報学類卒業．同年 (株) 日本タイムシェア入社．1994 年トヨタケーラム (株) 入社．1997 年 (株) ネットワーク応用通信研究所入社．オープンソースソフトウェアの開発に従事．プログラミング言語の設計と実装に興味を持つ．ACM 会員．

前田　敦司（正会員）

1994 年慶應義塾大学大学院理工学研究科数理科学専攻単位取得退学．博士（工学）（慶應義塾大学 1997 年）．1997 年電気通信大学大学院情報システム学研究科助手．2000 年筑波大学電子・情報工学系講師．2004 年筑波大学大学院システム情報工学研究科助教授（現職）．システムプログラム，プログラミング言語の実装，ガーベッジコレクションなどに興味を持つ．日本ソフトウエア科学会，ACM 各会員．

並木美太郎（正会員）

1984 年東京農工大学工学部数理情報工学科卒業．1986 年同大学大学院修士課程修了．同年 4 月（株）日立製作所基礎研究所入社．1988 年東京農工大学工学部数理情報工学科助手．1989 年電子情報工学科助手．1993 年 11 月電子情報工学科助教授．1998 年 4 月情報コミュニケーション工学科助教授．2007 年 4 月大学院共生科学研究院教授．博士 (工学)．オペレーティングシステム，言語処理系，ウィンドウシステムなどのシステムソフトウェア，並列処理，コンピュータネットワークおよびテキスト処理の研究・開発・教育に従事．ACM，IEEE，各会員．