Rubyのスレッド実装の改善

笹田 耕 —[†1]

プログラミング言語 Ruby はスレッドプログラミングをサポートする．スレッド処理機構を実現するため，Ruby 1.8 処理系はユーザレベルスレッドを独自に実装していたが，我々は最新の Ruby 1.9 処理系において，OS などが提供するネイティブスレッドを用いる方式で実装した．具体的には，POSIX Tthread（Pthread）と Windows スレッドに対応している．しかし，この Ruby 1.9 処理系のスレッド実装は次のような問題点がある．(1) タイマスレッドが定期的に監視を行うため、CPU を低消費電力状態に保つことができない (2) 複数 CPU での CPU 利用権の放棄がうまくいかない．我々はこれらの問題に対して，(1) タイマスレッドの改善 (2) CPU 利用権の受け渡し方式の変更，を行うことで問題点を解決し，スレッド実装を改善した．本稿では既存の Ruby 処理系でのスレッド実装について述べ，問題点をまとめ，この改善手法について述べる．そして，改善後の Ruby 処理系の性能について評価した結果を示す．

The Improvement of Thread Implementation for Ruby Interpreter

Koichi Sasada [†1]

Programming language Ruby supports thread programming. To support the threads, the Ruby 1.8 interpreter implements own userlevel threads. On the other hands, we made a thread mechanism using native threads provided by OSs for newest version of the Ruby interpreter Ruby 1.9. To be specific, POSIX Thread (Pthread) and Windows threads are supported. However, the Ruby 1.9 interpreter has several issues around threads implementation. (1) We can not keep CPU lowpower state because the timer thread wake up periodically (2) CPU utilization right can not be passed correctly on the SMP systems. We solve these issues by: (1) avoiding timer threads and (2) changing CPU utilization right passing method. In this paper, we show the current Ruby threads implementation, summerize issues and describe solutions. Moreover, we show the peformance of the modified Ruby interpreter.

1. はじめに

スクリプト言語 Ruby[4)–6)] 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[8)]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．とくに，Ruby on Rails[2)] などが有名であり，ウェブアプリケーション開発に多く利用されている．

Ruby は言語レベルでマルチスレッドプログラミングに対応している，という特長がある．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．スレッドを用いることで，非同期な I/O 処理など，平行性を有するプログラムを自然に作成することができる．Ruby ではブロックの記法を用いることで，スレッドを容易に作成することができる．

言語処理系にスレッドを実装する手法はいくつかあるが，最新版の Ruby 1.9 処理系では，OS 等が提供するネイティブスレッドを用いて実装している[7)]．具体的には，POSIX Thread[3)] と Windows スレッドに対応している．ここでは，OS が提供するスレッドと Ruby レベルで管理するスレッドを区別するため，前者をネイティブスレッド，後者を Ruby スレッドということにする．

最新版の Ruby 1.9.2 処理系が 2010 年 8 月にリリースされ，Ruby 1.9 系の版はより広く利用されるようになった．しかし，広く利用されることで，現在の Ruby スレッド処理機構に問題があることも，利用者からの報告などから判明してきた．本稿では，その中で 2 つの問題点に注目し，その改善について議論する．

まず，CPU の低消費電力機構を妨げるという問題である．最近の CPU および OS の機能に，利用効率の低いときに周波数を下げて消費電力を削減する機能がある．しかし，Ruby 1.9 でのスレッド実装では，タ

†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

イマスレッドという定期的に起動するヘルパスレッドを用いているため，周波数を下げたままにすることが出来ないという問題がある．

次に，マルチ CPU，マルチコア環境（SMP 環境）において，CPU 利用権を適切に他の Ruby スレッドに受け渡すことが出来ないという問題である．Ruby 1.9 処理系はネイティブスレッドを利用しているが，スレッドセーフでない C の関数などの存在を考慮して，スレッド切り替えのタイミングを抑制し，また並列計算機上で並列に実行しないようにするためジャイアントロック（GVL: Giant Virtual machine Lock，実行権ともいえる）を用いており，GVL を所有する Ruby スレッドのみに実行を許可している．プリエンプションは一定時間後，自発的に GVL を解放して実現する．しかし，SMP 環境下では GVL の受け渡しが正しく行われず，同じ Ruby スレッドのみが連続して GVL を獲得し，他の Ruby スレッドが実行できない，という問題がある．

これらの問題に対処するため，次のようなスレッド処理機構の改善を行った．まず，CPU の低消費電力機構を妨げる問題については，必要ない場合はタイマスレッドによる定期的な起動を行わないようにした．また，CPU 利用権（GVL）の受け渡しについては，GVL の受け渡しを，単純なロックの解放と獲得ではなく，SMP 環境で正しく受け渡しができるような実装へ変更した．

本研究の貢献は次の 2 つである．まず，Ruby 1.9 処理系を CPU の低消費電力機構や SMP 環境など，最近のハードウェアを活かすような機構へ，スレッド機構を改善したことである．これは，実世界で利用されるソフトウェアの性能向上を意味する実際的な貢献といえる．次に，ネイティブスレッドをユーザレベルスレッド機構で制御するための，現実的な環境での実現手法について議論している．通常，OS などが制御するネイティブスレッドを，さらにユーザレベルで管理するための手法について，Linux などの現実的な環境で行うための知見を述べる．

2. Ruby 1.9 のスレッドとその実装

本章では Ruby でのスレッドプログラミングと，Ruby 1.9 でのスレッド機構の実装について述べる．説明に必要な図や文章は，先行研究である文献 7) から適宜引用している．

なお，文献 7) では Ruby スレッドを並列に実行するための Ruby スレッド処理機構について述べているが，現在の Ruby 1.9 処理系は Ruby スレッドの並列実行は許可しておらず，GVL を用いて同時に実行する Ruby スレッドをたかだか 1 つに制限している．これは，スレッドセーフでない処理系や拡張ライブラリのプログラムを正しく実行するため，そのような制限を課している．本章では GVL についても触れる．

```
m = Mutex.new
th1 = 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

th2 = Thread.new do
  # (B) ここに記述した処理を新しいスレッドとして実行
  gets() # (B:io) I/O 処理
end

# (C) ここに記述した処理は (A), (B) 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
# ...
th1.join # ここでスレッド th1 と合流する
th2.join # ここでスレッド th2 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

2.1 Ruby スレッドを用いたプログラミング

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．`Thread.new` に渡した，`do ... end` で囲まれたブロックの中身 (A)，(B) と，その後ろに書かれた処理 (C) は別スレッドとして並行に実行される．

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 `th1`，`th2` 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．

`th1`，`th2`，および元々存在する Ruby スレッド（メインスレッド）は並行に実行されるが，並列には実行されない．ただし，B:io で示したような I/O 処理は，実行がブロックする可能性があるため，Ruby スレッドの実行とは並列に実行される．複数実行可能な Ruby スレッドが存在するときは，処理系に設定してあるタイムスライスごとに実行する Ruby スレッドを切り替える（プリエンプション）．

2.2 ネイティブスレッドを用いた Ruby スレッド処理機構の実装

Ruby1.9 処理系では 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している．**図 2** に Ruby 1.9 処理系の 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 2 の RT は Ruby スレッド，NT はネイティブス

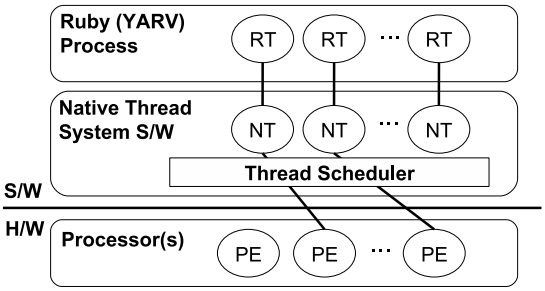

図 2 Ruby スレッドシステムの全体像

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，並行実行単位に割り付けられ，並行実行する．

2.2.1 ネイティブスレッドの利用方式

Ruby スレッドを実装するにあたり，OS などが提供するネイティブスレッドを利用している．これは，いくつか理由があるが，Ruby スレッドのコンテキストスイッチが高速である点，SMP 環境上でスレッドセーフな処理を並列実行させるため，そしてネイティブスレッドを利用している C 拡張などを行いやすい，といった理由で採用している．

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が，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ている．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．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，この 2 つであり，これに対応している．

2.2.2 Ruby スレッドの制御

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は，ネイティブスレッドの ID を示す値や，スレッドの状態を示す値，VM を実行するためのスタックなどが格納されている．

Ruby スレッド制御は仮想マシンの命令ではなく，対応する C で実装されたメソッド（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．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．開始するときには GVL を獲得して実行する．

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．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するようにして独自に実装した．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同期機構を組み合わせて実現している．割り込みにより中断しなければいけない可能性があるため，ネイティブスレッドの排他制御機構をそのまま利用できないためである．

2.2.3 Ruby スレッドのスケジューリングと GVL

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のスケジューラに任せる．コンテキスト切り換えは，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を期待している．

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．

Ruby スレッドが実行するためには，ジャイアントロック（GVL: Giant Virtual machine Lock）を獲得する必要がある．GVL の獲得と解放は，Pthread の `pthread_mutex_lock`，Windows での `CriticalSection` を用いて実装した．ブロックなどを行う処理や，スレッドセーフであることが保証できるプログラムを実行するとき，GVL を解放して実行する．これにより，それらの処理を Ruby スレッドの実行と並列に実行することができる．

2.2.4 割り込みとブロック解除関数

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．

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して行う．対象 Ruby 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．

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（以降，ブロッ

ク状態という），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．そこで，ブロック解除関数を登録する仕組みを設けている．ブロック解除関数は，ブロック要因に応じて，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り，ブロック状態になる可能性あるのある処理（例えば，`select` システムコールを含む処理）を行う前にブロック解除関数を登録し，処理を行ったあとにその登録を解除する．割り込み処理が行われた時には，対象とする Ruby スレッドに割り込みフラグを設定し，もしブロック解除関数が登録されていれば，その関数を呼び出し，ブロックの解除を行う．

なお，正しいブロック解除関数を設計・実装することは，意外と難しい．たとえば，`flock` システムコールを含む処理をキャンセルすることを考える．`flock` は他スレッドからその実行をキャンセルすることを考えて設計されていないため，唯一の方法はそのスレッドに対してシグナルを送信し，`errno` を `EINTR` として返ることを期待することになる．しかし，タイミングによっては `flock` を実行する直前にシグナルを送信することになる．この場合，その後に `flock` を行うことになり，キャンセルを正しく行うことができない．そのため，Ruby 1.9 処理系では，このような場合はブロック解除関数の登録が解除される，つまりブロック処理が終わるまで，定期的にシグナルを送信して確実にキャンセルできるようにしている．

2.2.5 タイマスレッドとシグナル

Ruby 1.9 処理系は他の Ruby スレッドに定期的に実行権を移すことによってプリエンプションを実現している．これは，前項で述べた割り込みフラグを定期的にセットすることにより実現している．

定期的に割り込みフラグを設定するために，Ruby 1.9 処理系はタイマスレッドというものを利用している．タイマスレッドは Ruby スレッドとは別のネイティブスレッドとして生成しており，実行中のスレッドの割り込みフラグにタイマイベントを設定し，一定時間スリープ（現在の実装では 10ms）して定期的なタイマイベントの生成を実現している．前項の最後で述べた，ブロック状態をキャンセルするまで定期的にシグナルを送信するのも，このタイマスレッドを用いて行う．

また，UNIX 等でのプロセス間通信で用いられるシグナルを，Ruby はサポートしており，シグナルを受信したときに実行する Ruby プログラムを設定できるが，この実現のためにもタイマスレッドを用いている．

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設定することができるが，シグナルはマスクされていないどこかのネイティブスレッドに配送される．シグナルを受信すると，前述のようにシステムコールなどはキャンセルされるため，キャンセルに対処していないようなシステムコールの呼び出しでは，意図しない挙動となる可能性がある．そこで，タイマスレッドのみシグナルマスクをしないようにし，シグナルはタイマスレッドに集約するようにしている．なお，シグナルを受信したら，メインスレッドへ割り込みを発生し，Ruby レベルでのシグナル処理を行うようにしている．

2.3 Ruby 1.9 のスレッド実装の問題点

Ruby 1.9 処理系での Ruby スレッド処理機構の問題点が，すでにいくつか指摘されている．本稿ではその中で 2 つについて着目する．本節ではその 2 つの問題点と，さらに本稿では扱わない問題を 2 つ紹介する．

2.3.1 CPU の消費電力機構の阻害

最近の CPU には，実行状況に応じて動作周波数を変更し，消費電力を削減するための機能がある．対応している OS は，システム全体の CPU 利用率を測定し，利用効率が低い判断すると動作周波数を下げ，システム全体の消費電力を削減する．

しかし，Ruby 1.9 処理系では，タイマスレッドを必ず設定し，10ms に一度必ずタイマスレッドの処理が走る．どの Ruby スレッドも実行していない状況でも，この処理が走るため，OS はシステムの使用率が低いと判断することができず，CPU を低周波数の状態に保つことが出来ない．

この問題は ruby-core という Ruby 開発メーリングリスト（英語）で，たびたび指摘されており，修正案およびその実装も提案された．ただ，その修正案は Ruby スレッドが 2 つ以上存在する場合にタイマスレッドを生成する，というものであり，2 スレッド以上が I/O 処理でブロックした場合や，定期的なシグナル送信のための機能などがおざなりになったものであり，受け入れられるものではなかった．

2.3.2 SMP 環境での GVL 受け渡し

マルチ CPU，マルチコア環境（SMP 環境）において，GVL を受け渡すことが出来ない，という問題点が存在した．具体的には，Ruby スレッドを受け持つネイティブスレッドが，異なる CPU コアに割り付けられている場合，ある Ruby スレッドが GVL を解放しても，他の Ruby スレッドが獲得するより前に，また同じ Ruby スレッドが GVL を獲得して，GVL が永遠に受け渡らない，という状況が発生してしまった．

例えば，**図 3** では，親スレッドは子スレッドが `flag` という変数が偽となるまで繰り返すプログラムだが，この問題により GVL が子スレッドに受け渡すことができず，無限ループとなってしまう，という問題である．

この問題は，GVL の実装に単純なロックを利用しているために生じる（Pthread 環境では `pthrea_mutex_t`，Windows の環境では `CRITICAL_SECTION`）．これらの API は，ロック解放後，その他のネイティブスレッドが待っていても，そのネイティブスレッドに必ずロック獲得権が渡るということを保証していないためで

```
flag = true
Thread.new{flag = false}
while flag
  # infinite loop
end
```
図 3 止まらない可能性のある Ruby プログラムの例

ある．

この理由を推測すると，次のようになる．これらのプリミティブなロック機構は，出来るだけ少ない実行コストであることが求められている．必ず待っているネイティブスレッドにロック獲得権を渡すことを保証するためには，SMP 環境で異なる CPU コアに属するネイティブスレッドのスケジューリングに介入する必要があり，大きな実行コストがかかる．そのため，これらの API を利用する限り，確実な GVL の受け渡しを行うことが出来ない．

2.3.3 その他の問題点

本稿では扱わないが，問題であろうと思われる点について補足する．

現在の Ruby 1.9 の実装では我々の先行研究である文献 7) で述べたような並列実行する Ruby スレッドを実現していない．つまり，SMP 環境における Ruby スレッドの並列実行はサポートしていない．

文献 7) で述べたとおり，文字列などを含む処理では結局 GVL を獲得せねばならず，逆に C メソッドの実行ごとに GVL の獲得が必要となり，他のネイティブスレッドとロックの競合を発生することで全体的なパフォーマンスの低下が見られた．これを解決するためにはスレッドセーフで GVL の不要な C メソッドを増やす必要があるが，実際に開発する人間のマンパワー的な問題として，これをバグの無い品質でリリースすることは難しい．これらの判断から，並列実行する Ruby スレッドは採用していない．

この問題については，そもそも本当に Ruby スレッドを並列に実行させる機構が Ruby プログラミングに必要か，という懸念がある．並列なスレッド実装を提供する Java などのプログラミング言語において，排他制御にミスがあると容易にバグを発生させるなど，困難なことが知られている．スクリプト言語は容易なプログラミングを行うための言語という側面が強いため，プログラミングを難しくするような Ruby スレッドの並列実行は，他の並列実行のための仕組みはないかなど，十分検討しなければならない．

ネイティブスレッドを利用する問題点であるが，一部の UNIX 系 OS において，`fork` と Pthread との相性によりフリーズする，という問題点が知られている．具体的には，FreeBSD や NetBSD の古いバージョンで，一度でも Pthread によってネイティブスレッドを生成したプロセスが `fork` によって子プロセスを生成し，子プロセスで `exec` などをしないで新たにネイティブスレッドを生成しようとするとプロセスがフリーズしてしまう，という問題である．

POSIX では，`fork` 後に `exec` 以外のシステムコールを実行することは未定義なので，正しい仕様といえる．しかし，Ruby 1.9 処理系では `fork` で生成された子プロセスで，まずタイマスレッドを生成するため，その時点でプロセスがフリーズしてしまう．

この問題に対処するためには，タイマスレッドのようなスレッドを作らない，もしくは必要になるまで生成を遅延する，という解決策が考えられるが，問題となる OS の最新バージョンではこの問題が解決しているため，この問題への対処はしないこととした．

3. 問題点の解決

本章では，前述した 2 つの問題の解決方法を検討し，その実装を述べる．本章では一般的な方式というよりも，Linux などの API を利用して，現実的な実装方法を探るという視点で述べている．

また，副次的に改善することができたシグナルの取り扱いについても紹介する．

3.1 タイマスレッドの改善

前章の最後にまとめたように，タイマスレッドが定期的（10ms ごと）に起動してしまうため，CPU を低消費電力状態に保つことができない，ということであった．そこで，GVL を手放す必要があるときのみ，起動するように変更した．

本節では，Pthread 環境での対応について述べる．Windows については，執筆時現在，まだ対応できていが，同様の方式により実装可能と考えている．

3.2 検　　討

タイマスレッドが定期的に動作しないようにするために，2 つの方式を検討した．

必要なときのみタイマスレッドを起動する方式　タイマスレッドが必要なのは，1 つ以上の Ruby スレッドが GVL 獲得を待っている状態である．そのため，従来のタイマスレッドを変更し，もしどの Ruby スレッドも GVL を待っていなければ，定期的な起動を実行しないようにする．

`setitimer` を用いる方式　POSIX には `setitimer` という，定期的にシグナル（`SIGVTALRM`）を配送するための機能がある．この機能を用いて，待ち GVL がいる状態では定期的にシグナルを配送するようにして，シグナルハンドラにてタイマの割り込みフラグを設定する．

直感的には前者の修正が簡単である．しかし，前者は実行中の Ruby スレッドに加えてタイマスレッドがやはり起動することになる．つまり，2 コアの計算機の場合，前者では 1 コアのみが実行状態で，残りの 1 コアがアイドル状態であるはずが，タイマスレッドを用いる限り，2 コア目も定期的に実行する，という元の問題が発生してしまう．

しかし，実際に実装してシグナル通知タイミングを計測してみると，一部の環境では `setitimer` による `SIGVTALRM` の配送タイミングは一定ではなく，定期的な時間計測の用には向かないと言うことがわかった．また，すでに 1 つ以上のコアが動作中に，もう一つのコアがタイマスレッドによって起動されることによる消費電力の増加は，どのコアも実行状態でない時に起動するよりも相対的に少ない．これらの理由から前者を採用することにした．

3.3 実　　装

GVL 獲得を待つスレッドの数を管理し，GVL 獲得待ちスレッド数がある場合のみ，タイマスレッドを有効にすることにした．ある Ruby スレッドが GVL を獲得しようとするとき，すでに他の Ruby スレッドが GVL を獲得していれば，その Ruby スレッドは GVL 獲得待ちとなる．このとき，GVL 獲得待ち Ruby スレッド数が 0 であれば，タイマスレッドはタイマ機能を停止しているため，タイマスレッドに定期的に実行中の Ruby スレッドにタイマ割り込みをかけるようにする．このタイマスレッドへの通信にはセマフォを用いた[⋆1]．

タイマスレッドは，タイマ割り込みが必要な場合，もしくは 2 章で述べた連続してシグナルを送るような場合には，自身を定期的に起動するように設定し（`sem_timedwait` を用いた）実行を一時停止する．もし，不要な場合は他のネイティブスレッドから通信が入るまで停止するようにした（`sem_wait` を用いた）．

3.4 GVL 実装方式の改善

前章において，GVL の獲得，解放を既存の低レベルロック機構を用いると問題であることを示した．そこで，GVL の実装手法を変更することでこの問題を解決する．

Windows の場合は `CRITICAL_SECTION` ではなく，`Mutex` オブジェクトを用いることで，GVL が他のスレッドに確実に受け渡しが出来ることを確認した．これは，`Mutex` オブジェクトが待ちスレッドの FIFO を内部で実装しており，必ず最初に待っているオブジェクトにスケジューリングが渡ることを保証しているためだと思われる．

Pthread 環境では，Windows における `Mutex` オブジェクトのようなものが存在しなかったため，独自に実装した．ただし，この方式にはいくつか考えられるため，以降本節では Pthread 環境における GVL の実装方式について詳しく述べる．

3.4.1 検　　討

まず，`pthread_mutex_t` を用いるのではなく，セマフォや条件変数を 1 つ用いて GVL を実装してみたが，どれも同様の問題が発生することを確認した．どの同期インターフェースも，待っているネイティブスレッドに確実にロックを受け渡すことは保証していないためである．

そこで，次のような方式を検討した[⋆2]

ecv　すべての Ruby スレッドに条件変数を持たせ，GVL を獲得するときにはその条件変数で待機することにする方式．どの Ruby スレッドを起こすかを，GVL 解放時に柔軟に決定することが出来る．今回は，独自に FIFO による待ち Ruby スレッドリストを作成した．すべての Ruby スレッドが条件変数をもつため ecv という略語とする．

2cv　GVL 解放時，他に GVL を待つ Ruby スレッドがいれば，他の Ruby スレッドが GVL 獲得するまで待機する方式．他の Ruby スレッドが GVL 獲得するまで待つため，連続して GVL を獲得するようなことはしない．GVL 解放を待つための条件変数と，GVL 獲得を待つための条件変数があるため，2cv という略語とする．Python の開発中のバージョンで，この方式を利用している．

2lock　2 つのロックを用いて確実に GVL を受け渡す方式．ロックを 2 つ（L1，L2）用意する．GVL 獲得時，まず L2 を獲得し，L1 を獲得する．ロック獲得後，L2 を解放して実行を再開する．GVL 解放時，L1 を解放する．この方式では，GVL 獲得待ちの Ruby スレッドがいた場合，その Ruby スレッドが L2 を獲得しているため，GVL を解放した Ruby スレッドが連続して GVL を獲得するようなことは起こらなくなる．

al　最後に GVL を獲得したスレッドは必ず条件変数で待つ方式．GVL を最後に獲得した Ruby スレッドを記録しておき，もし GVL 待ち Ruby スレッドがいた場合は必ず GVL 解放通知を受け取る条件変数で待つ，という実装である．al は avoid last という意味で名付けた．

これらの方式は，すべて正しく GVL の受け渡しができていることを確認した．

次に，GVL の獲得，解放を複数スレッドで連続して行うベンチマークプログラムでこの性能を調べた．結果として，2lock がもっとも良い性能を示した．これは，条件変数よりも，単純なロックを用いるためであると思われる．ecv は最も悪い性能を示した．

ただし，どの方式も，既存の単純なロックを用いた GVL よりも低い性能となることがわかった．これは，

⋆1 MacOS X ではプロセスローカルなセマフォに対応していないようなので，原稿執筆時には既存の実装のままとするようにしている．

⋆2 ここで述べた改善と議論は，所属の異なる Ruby 開発者（謝辞に詳しい）間で IRC を用いて進められた．ecv 以外のアイデアは筆者ではなく，他の Ruby 開発者からの提案であることを明記しておく．議論が実装とデータを交えて，数日間，深夜から朝にかけて活発に交わされた．このような場所と組織を超えた議論・開発の進め方は，オープンソースソフトウェア（OSS）開発ならではと言えるかもしれない．

I/O を多発するようなアプリケーションで問題になる．I/O，つまりブロッキングを伴うような処理では，GVL を解放し，処理後に再獲得する．このような解放，獲得が頻発すると，この変更により性能を落とす可能性がある．

この性能低下の原因は，CPU コアをまたいだ GVL の移送が，CPU コアをまたいだネイティブスレッドのスケジューリングを発生させるためだと考えられる．ネイティブスレッドは OS などがスケジューリングするが，そこからさらにユーザレベルで無理矢理再スケジューリングしていることの弊害であると考えられる．

そこで，I/O などで一時的に GVL の解放，獲得を行う場合と，タイムスライスが切れて GVL を他の Ruby スレッドに渡す場合を明確に分けることにした．前者は，同じ Ruby スレッドの GVL 再獲得を許す（むしろ，性能的には好ましい）ように実装し，後者は必ず GVL を受け渡すようにした．

具体的には，上記 al 手法を拡張し，I/O などで一時的な GVL の解放，獲得時には同一スレッドチェックをしないように変更した．この方式を al2 とする．

3.4.2 議　　論

性能的には al2 がもっとも良い性能を示したため，この方式を採用しようと検討している．ただし，スケジューリングの柔軟性という点では ecv が最もよく，優先度付きスケジューリングなどがユーザレベルで実装可能である．

本稿執筆時には間に合わなかったが，ecv に al2 のような，一時的な GVL の解放・獲得時には厳密なスケジューリングをスキップする機構を用いるのはどうかと考えている．実現すれば，Ruby レベルでのシグナルハンドラを実行する Ruby スレッドを最優先にスケジューリングし，シグナルへの反応速度を向上することができる．

3.5 シグナルマスクの改善

今回の 2 つの問題の解決の副産物として，シグナルマスクに関する改善を行ったので，これについても報告しておく．

3.5.1 新しいシグナル配送の実装

2.2.5 で述べたように，Ruby 1.9 処理系ではタイマスレッドにシグナルを集約し，さらにメインスレッドへ配送する，という手順となっていた．これは，任意の（Ruby スレッドを含む）ネイティブスレッドがシグナルによって中断されるとまずい，と考えたためである．

しかし，実は Ruby 1.9 処理系は，バグのため，長らく適切なシグナルマスクをかけておらず，どの Ruby スレッドもシグナルのために中断しえる状態だったということである．これに関するバグの報告は受けていないため，つまり，心配は杞憂であるという可能性が大きいことがわかった．

そこで，方針を変更し，とくにシグナルマスクを設定することはなく，どのネイティブスレッドもシグナルを受けることを出来るようにした．`signal` システムコールによって設定したプロセス共通のシグナルハンドラは，まずタイマスレッドに OS からのシグナル配送を通知する．タイマスレッドは，メインスレッドへシグナル配送があったことを割り込みによって通知する．

シグナルハンドラからタイマスレッドへの通知は，非同期シグナル安全な操作であるセマフォにより実装した．

3.5.2 `EINTR` への対応のチェック

前項では，シグナルに正しく対応できている，という根拠を，これまでバグ報告がないため，とした．しかし，非同期シグナルはタイミングの問題が大きいため，問題が潜在している可能性は大きい．そこで，`ptrace` システムコールを利用して，すべての `EINTR` を返す可能性のあるシステムコールで `EINTR` を発生し，正しく処理できることを確認することにした．

`ptrace` システムコールを利用すると，システムコールの呼び出し，戻り値を監視，もしくは書き換えを行うことができる．そこで，システムコール呼び出しが `EINTR` によって 100 回失敗し，その後成功するようにした．正しく処理している場合は，`EINTR` は単に再試行するだけであるはずである．

この方式を用いて Ruby 1.9 処理系付属のテストセットを Linux 上で実行したところ，1 箇所のチェック漏れを発見した．逆に言うと，1 箇所以外は，正しく対処していることを確認できた．

4. 評　　価

本章では，改善を行った Ruby 処理系を用いて，電力消費量の改善効果と，実行性能を評価した結果について述べる．

4.1 電力消費量の削減

電力消費量を計測するために，Watts Up[1)] という電力計を利用した．この電力計は，単位時間（ここでは 1 秒）あたりの計測結果をロギングし，記録したログを USB 経由で PC に転送することができる．この電力計をもちいて，アイドル状態の Ruby プロセスを 2^n（$n = 0 \ldots 7$）個同時に立ち上げ，60 秒間実行したときの電力消費量を計測した．具体的には，各 Ruby プロセスは `sleep` メソッドによってアイドル状態を保つ．アイドル状態のプロセスをいくつ同時に立ち上げたところで，電力消費量は増えない，というのが理想である．

評価環境は CPU が Intel Core i5，OS は Ubuntu 10.10 をデフォルトのまま利用している．改良前 Ruby は ruby 1.9.2p0 (2010-08-18 revision 29036) [x86_64-linux]，改良版 Ruby は ruby 1.9.3dev (2010-12-07 trunk 30123) [x86_64-linux] をベースに用いた．

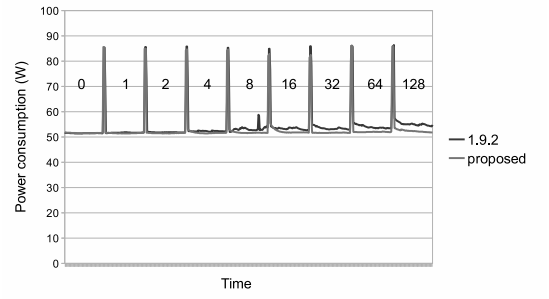

図 4 消費電力の評価（0〜128 プロセス同時起動時の電力比較）

評価結果を**図 4** に示す．グラフ中の 0〜128 の数字が同時に実行しているアイドル状態の Ruby プロセスの数である．プロセス数を変更する時に 3 秒間，ビジーループによって負荷をかけ消費電力を増やし，プロセス数の区切りをわかりやすくしている．

結果を見ると，たしかに Ruby 1.9.2 はプロセス数が増えると，アイドル状態にもかかわらず消費電力が増えている．例えば，0 プロセスから 1 プロセスに増えると約 0.3W の消費電力増加が観測できる．しかし，提案手法ではアイドル状態のプロセス数が増えても消費電力が増えないことがわかる[⋆1]．

ただ，そもそもこの計算機システムは，アイドル状態でも 50W 強消費しているため，提案自体の消費電力削減効果は少ない．この成果は，より消費電力の低い計算機上にて活かせるのではないかと考えられる．

4.2 性 能 評 価

Ruby 1.9 処理系付属のベンチマークセットを用いて，前節と同じ環境で性能評価を行った（比較対象は改善後のものと，改善後のものを GVL の実装について単純なロックに戻したもの）．

その結果，ほとんどのベンチマーク結果に変化はなかった．これは，ほとんどのベンチマークがスレッドを用いておらず，スレッドを用いていないプログラムへの影響が出ないような実装となったということである．

しかし，多スレッドで Ruby レベルの排他制御を行うベンチマークにおいて，約 800 倍遅いという異常な結果が出た．これは，GVL のアルゴリズム変更に排他制御機構を追従させなければならないところ，出来ていないことを示している．執筆時点では間に合わなかったが，発表時点ではこの点も解消することを目指したい．

5. ま と め

本稿では，Ruby 処理系の改善について 2 例紹介した．1 つ目は CPU を低消費電力状態を阻害するタイマスレッドを改善したこと，2 つ目は SMP 環境におけるジャイアントロックの受け渡しを確実に行うようにしたことである．ジャイアントロックの受け渡しについては，CPU コア間をまたがるスケジューリングが発生すると低い性能となることがわかったため，それをなくべく起こさないような方式を検討した．評価の結果，ごくわずかだが電力消費量が低下し，性能低下はほぼ見られなかった．しかし，排他制御機構がまだ追従できておらず，性能が非常におちていることがわかった．これはすぐにでも解決するべき課題であり，発表までには正しく対処したいと思う．

一般的なスケジューリングではなく，ネイティブスレッドを用いるような処理系が，処理系自身のスケジューリングをどのように行うべきか，というあまり一般的ではないが，現実的な問題について検討した．本稿が実際のソフトウェア開発の一助になれば幸いである．

謝　　辞

本稿での議論は，Ruby 開発コミュニティの助言を受けました．とくに，GVL の実装方式については，連日深夜の議論に参加して下さった小崎資広氏（富士通），樽家昌也氏（(株) 東芝 研究開発センター コンピュータアーキテクチャ・セキュリティラボラトリー），遠藤侑介氏（(株) 東芝 研究開発センター システム技術ラボラトリー）に，方式や実装を提案して頂きました．改めて感謝致します．

本研究の一部は科研費（21700024）の助成を受けたものです．

参 考 文 献

1) : Watts up? PRO ES. https://www.wattsupmeters.com/secure/products.php?pn=0.
2) Hansson, D.H.: Ruby on Rails: Web Development that doesn't hurt. http://www.rubyonrails.org.
3) IEEE: *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1* (1996).
4) Thomas, D., Fowler, C. and Hunt, A.: *Programming Ruby*, The Pragmatic Programmers (2004).
5) 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).
6) 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/.
7) 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 における並列実行スレッドの実装，情報処理学会論文誌. プログラミング， Vol.48, No.10, pp.1–16 (2007-06-15).
8) 松本行弘：Ruby の真実，情報処理， Vol. 44, No.5, pp.515–521 (2003).

⋆1 128 プロセス起動したとき，電力消費が収束するまで時間がかかっているが，これはプロセス生成のための OS の負荷だと思われる．