計算機アーキテクチャ　99－8
(1993. 3. 12)

Schemeにおけるユーザレベルのスレッド管理方式

戸川 敦之† 大久保 英嗣† 大野 豊† 白川 洋充††

† 立命館大学 理工学部 情報工学科
603 京都市北区等寺院北町 56-1

†† 近畿大学 理工学部 経営工学科
577 東大阪市小若江町 3-4-1

あらまし

より軽量なプロセスを実現する必要から，多くのオペレーティングシステムでは，軽量プロセスを管理する機能をカーネルが提供している．しかし，この方法では，カーネルとユーザプロセス間のコンテキストスイッチに伴うオーバヘッドが無視できない．本論文ではScheme言語から直接ユーザレベルでスレッドの管理を行うライブラリの実現法と，このライブラリを効率的に実行できる処理系を実現する手法について述べる．

このライブラリの評価を行った結果，Mach3.0が提供する軽量プロセスを上回る性能を持ち．また，コルーチンによって実現されたC言語のためのライブラリと性能において有意な差が無いことがわかった．

和文キーワード　第一級の継続 ユーザレベルのスレッド管理 Scheme言語

User-Level Management in Scheme Threads

Atsushi Togawa† Eiji Okubo† Yutaka Ohno† Hiromitsu Shirakawa††

†Department of Computer Science and Systems Engineering,
Faculty of Science and Engineering,
Ritsumeikan University
56-1 Tojiin, Kita-ku, Kyoto 603, Japan

††Department of Industrial Engineering,
Faculty of Science and Engineering,
Kinki University
3-4-1 Kowakae, Higashi-Osaka 577, Japan

Abstract

Many operating systems provide the facility of light-weight processes to increase efficiency. But efficiency of the method is not sufficient since the overhead of context switching between user spaces and a kernel space is not so small.

In this paper, the implementation of a light-weight process library for Scheme Threads and its runtime library is presented.

Performance results of the Scheme Threads show that it is faster than C Threads and it is comparable to C Threads coroutines.

英文 key words　first-class continuation, thread management at user level, Scheme

1 はじめに

より軽量なプロセスを実現する必要から, 多くのオペレーティングシステムでは, 軽量プロセスを管理する機能をカーネルが提供している. しかし, この方法では, カーネルとユーザプロセス間のコンテキストスイッチに伴うオーバヘッドが無視できない. そのため, プロセス生成やプロセス切り替えのオーバヘッドはさほど軽減されない. この問題を解決するために, ユーザレベルでスレッドライブラリを実現し, カーネルとアプリケーションプログラム間の制御の移行に起因するオーバヘッドを解消することが試みられている.

我々はScheme言語のためのユーザレベルスレッドライブラリであるScheme Threadsを開発した. このライブラリはC Threads[3]が持つ種々の機能を備えている. その実現にはScheme言語の特徴の一つである第一級の継続を利用している.

本論文では, 2章でScheme言語が持つ第一級の継続について述べた後に, 3章でScheme Threadsが備えているインタフェースを概説する. そして, 4章で第一級の継続を用いてスレッドのコンテキストを表現する手法について述べる. 5章でScheme Threadsを効率的に動作させるための処理系の実現手法を考察し, 6章でその性能の評価を行う.

2 第一級の継続

継続(continuation)とは, 式が評価結果を受け取り, 残りの計算を行う関数である. つまり, 継続はある時点以降の計算を表現している関数であるといえる. 継続をプログラマが操作することによって様々な制御構造を実現することができる.

Scheme言語が持つ基本手続き `call-with-current-continuation`(以後, `call/cc`と省略する)によって, エスケープ手続きと呼ばれる特殊な手続きが得られる. エスケープ手続きを呼び出すと, 呼び出した時点の継続は破棄され, `call/cc`を呼び出したときの継続へと制御が移行する.

例えば, 以下に示すプログラムでは, 手続き `f` の呼び出しも, その後の `saved-k` の呼び出しもともに `f` 中の文字列"string"を結果として返す.

```
(define (f)
   (call-with-current-continuation
    (lambda (k) (set! saved-k k) ))
   "string")
(f) ; 手続き f の呼び出し
(saved-k #f) ; エスケープ手続きの呼び出し
```

上の例のように, `call/cc` はC言語の `longjmp` などのような大域脱出のための基本操作と異り, 手続きの内から外への制御の移行だけでなく, 既に実行が終了した手続き内への制御の移行も可能である.

3 Scheme Threads

3.1 スレッドの生成とスレッドの実行の制御

スレッドの生成とその活動を終了させるために以下に示す基本手続きを備えている.

```
(thread-fork procedure argument priority)
(thread-join thread)
(thread-exit result)
```

`thread-fork` を呼び出すことによって優先順位として `priority` を持つ新たなスレッドが生成され, このスレッド上で `argument` に対して手続き `procedure` が適用される. このようにして生成されたスレッドは以下の場合に活動を停止する.

1. 手続き `procedure` の評価が終了したとき
2. 手続き `procedure` の評価中に `thread-exit` が呼び出されたとき

スレッドは終了する際にスレッド終了ステータスと呼ばれる値を返すことができる. 1の場合では手続きの評価結果がスレッド終了ステータスとなる. 2の場合では `thread-exit` の呼び出し時に与えられた引数がスレッド終了ステータスとなる. 他のスレッドは `thread-join` を呼び出すことによってこの値を参照できる. この手続きを呼び出すと, `thread-join` の引数として与えられたスレッドが終了するまで処理が中断される. そして, このスレッドの終了ステータスが `thread-join` の戻り値として返される.

他のスレッドの実行を制御するために以下に示す手続きを用意している.

```
(thread-suspend thread)
(thread-resume thread)
(thread-set-continuation! thread k)
```

`thread-suspend` によって引数 `thread` で指定されたスレッドの実行を停止させることができる. このようにして停止したスレッドのコンテキストを `thread-set-continuation!` 手続きを用いて設定することができる. `thread-resume` を呼び出すことによりスレッドの実行が再開される.

これらの手続きの他に, 明示的にプロセッサを他のスレッドに明け渡す手続きとして thread-yield を用意している.

3.2 排他制御

排他制御を実現するための基本操作として以下に示す手続きを用意している.

```
(mutex-alloc)
(mutex-lock mutex)
(mutex-unlock mutex)
```

mutex-alloc は新たに mutex 変数を割り当てる手続きである. この手続きを呼び出すと結果として mutex 変数が得られる. mutex の獲得と解放はそれぞれ mutex-lock と mutex-unlock 手続きを呼び出すことにより行われる. 引数 mutex に mutex 変数を与えることにより, それぞれロックの獲得・解放が行われる.

3.3 同期制御

スレッド間の同期を実現するための基本操作として, 以下に示す手続きを用意している.

```
(condition-alloc mutex)
(condition-wait condition)
(condition-signal condition)
(condition-broadcast condition)
```

condition-alloc は条件変数を割り当てる手続きである. この手続きの呼び出し結果として, 新たに生成された条件変数が得られる. condition-wait は引数として与えられた条件変数 condition に対して condition-signal, condition-broadcast が実行されるまでスレッドの実行を停止する手続きである. condition-signal は複数のスレッドが待ち状態にあるとき, 一つのスレッドだけを実行可能状態に変える. これに対して condition-broadcast は, 待っている全てのスレッドを実行可能状態に変える.

3.4 スレッドデータの操作

スレッドデータは各スレッド毎に割り当てられた変数である. この変数に対する基本操作として以下に示す手続きを用意している.

```
(thread-data thread)
(thread-set-data! thread data)
```

thread-set-data! 手続きによってこの変数に対して任意の値を代入することができる. また, thread-data 手続きによってこの変数の値を参照できる.

3.5 call/cc の意味

Scheme 言語に並行性を導入する際に, call/cc 手続きの意味をどのように定義するかが問題となる. 例えば, エスケープ手続きは, call/cc を呼び出したスレッドが将来行う計算だけに関するものなのか, あるいは, そのスレッドが生成した全スレッドが行う計算も含むのかを定義しなくてはならない.

Scheme Threads では以下に示すように call/cc の意味を定義している.

- あるスレッドが call/cc によって得たエスケープ手続きをそのスレッドが呼び出すと, call/cc 以降の時点に制御が移行する. 他のスレッドには影響を与えない.
- あるスレッドが call/cc によって得たエスケープ手続きを他のスレッドが呼び出すと, このスレッドの制御が call/cc 以降の時点に移行する. エスケープ手続きを呼び出したスレッド以外のスレッドには影響を与えない.

このように定義することにより, call/cc の意味を大きく変更することを避けている.

4 継続によるコンテキストの表現

一般に, スレッドを管理するプログラムはスレッドの状態を保持するデータ構造を管理しているが, この状態とはスレッドが中断された時点の継続に他ならない. 従って, 第一級の継続を有する言語では, プロセスの状態を継続によって表現することができる [12].

Scheme Threads でスレッドの切替えの際に呼び出される手続き threads::dispatch と threads::block を図1に示す.

threads::dispatch 手続き中では, まず, call/cc を呼び出すことによって現在実行中のスレッドのコンテキストを含むエスケープ手続きが変数 k に代入される. 各スレッド毎に TCB(task control block) と呼ばれる配列が割り当てられる. threads::set-thread-context! は, TCB に call/cc によって得られたエスケープ手続きを格納する手続きである. 最後に, threads::block を呼び出すことによって, 実行可能なスレッドの中で最も優先順位の高いスレッドの実行を開始する.

threads::block 手続き中では, 最も優先順位の高い実行可能スレッドが threads::choose-thread 手続きによって選択される. threads::thread-continuation 手続きを呼び出すことによってこのスレッドの TCB からエス

```
(define (threads::dispatch)
  (call-with-current-continuation
   (lambda (k)
     (threads::set-thread-context! threads::current-thread k)
     (threads::block) )))

(define (threads::block)
  (cond ((threads::choose-thread) ; 実行可能なスレッド中で最も優先順位の高いスレッドを選択する
         => (lambda (new-thread)
              (set! threads::current-thread new-thread)
              ; コンテキストを表現するエスケープ手続きに対して, 引数 #fを与えて呼び出す.
              ((threads::thread-continuation new-thread) #f) ))
        (else (threads::block)) ))
```

図 1　継続によるコンテキストの表現

ケープ手続きが取り出される. このエスケープ手続きに引数 #fを与えて呼び出すことによって, threads::dispatch 中の call/cc の後へ制御が移行する.

5　処理系の実現手法

5.1　第一級の継続の実現

第一級の継続を実現する手法としていくつかの手法が知られている [1]. 本節ではこれらを比較し, Scheme Threads の動作に適した手法を考察する.

ガーベッジコレクション法　すべてのアクティベーションレコードをヒープ領域上に割り当て, ガーベッジコレクタによって解放を行う手法である. call/cc が呼び出されると, 呼び出した手続きのアクティベーションレコードと call/cc 手続きからの戻り番地からエスケープ手続きが構築される. エスケープ手続きが呼び出されると, call/cc を呼び出した手続きのアクティベーションレコードが現在のレコードとなり, call/cc からの戻り番地へと制御が移行する.

ヒープ法　アクティベーションレコードはヒープ領域に割り当てられる. 各レコードにはエスケープ手続きから参照されるか否かを示すビットがある. 手続きから戻る際にこのビットが立てられていなかった場合はこのレコードを解放するが, 立てられていた場合は解放しない. call/cc が呼び出されると, 現在のレコードから参照されるすべてのレコードのビットを立てる操作を行う. そして, call/cc 手続きからの戻り番地と call/cc を呼び出した手続きのレコードへのポインタからエスケープ手続きが構築される.

スタック法　手続きの呼び出しと手続きからの戻りは C 言語のようにスタックにアクティベーションレコードを割り当てる言語と同様の処理を行う. call/cc が呼び出されると, スタックの内容がヒープ領域に退避され, call/cc 手続きからの戻り番地とヒープ領域に格納されたスタックの内容へのポインタからエスケープ手続きが構築される. エスケープ手続きが呼び出されると退避されていたスタックの内容が復元され, call/cc からの戻り番地へと制御が移行する.

スタック / ヒープ法　アクティベーションレコードはスタック上に割り当てられる. call/cc が呼び出されると現在のスタックの内容がヒープ領域に退避され, スタックが空にされる. そして, call/cc 手続きからの戻り番地と退避されたスタックの内容へのポインタからエスケープ手続きが構築される. エスケープ手続きが呼び出されると, 現在のアクティベーションレコードをヒープ領域に退避されたレコードとし, スタックを空にする. 手続きからの戻りでは, 戻り先の手続きのアクティベーションレコードがスタック上にあるか, あるいはヒープ領域にあるかを判別し, それぞれに対する処理を行う.

インクリメンタルスタック / ヒープ法　スタック / ヒープ法とほぼ同様の手法である. 手続きから戻る際に, 戻り先の手続きのアクティベーションレコードがヒープ領域中にあった場合は, そのレコードと, そのレコードとともに退避されたレコードをスタックにコピーする点が異なっている.

これらの手法のうち, call/cc の実行とエスケープ手続きの呼出しが一定の時間で行えるのは, ガーベッジコレクション法のみである. そこで本処理系では, この手法を用いて第一級の継続を実現した.

```
(define (top-level-procedure)
  (print "hello."))

(define (value-holder value)
  (define (put new-value)
    (set! value new-value))
  (define (get) value)

  (cons put get))
```

図 2　Scheme が持つ手続きの種別

5.2　手続きの型の識別

Scheme では, 以下に示すようにいくつかの種別の手続きを扱う必要がある.

- トップレベルで定義された手続き
- 閉包
- エスケープ手続き

一般に, 手続きの呼び出しがどの種の手続きの呼び出しなのかをコンパイル時に決定することはできない. そのため, 実行時に手続きの種別を判別する必要がある.

トップレベルで定義された手続きとは, 図2の `top-level-procedure` や `value-holder` のように, その定義の外側を囲む手続きが存在しない手続きである. この種の手続きは, その実行開始番地を知るだけで正しく呼び出すことができる.

手続き `value-holder` は, その中で定義されている二つの手続き `get` と `put` を結果として返す. これらの手続きは `value-holder` の局所変数を参照している. 局所変数の実体の位置は `value-holder` の呼び出し毎に異なる. そのため, `get` 手続きや `put` 手続きを単なるプログラム実行開始番地として表現することはできず, プログラムの実行開始番地とともに `value-holder` のアクティベーションレコードも保持しなくてはならない. このようなデータは閉包と呼ばれる. 閉包を呼び出す時には適切なアクティベーションレコードを参照できるように, 現在のアクティベーションレコードの位置を保持するレジスタを設定した後に, 手続きのコードへ制御を移す必要がある.

これらの種別を呼び出し時に判別するために以下に示す方法が用いられる.

タグ法　Scheme では一般に変数や配列などが保持するデータの型は実行時まで定まらない. そのため, 一語に収容できない値は, 実際の値が格納された領域へのポインタをデータの値として扱い, 常に値を一語で表現する手法がとられる. さらに, この語の中にデータの型を識別するタグを格納することが多い. 異なる種別の手続きに異なるタグ値を与えることによって, 呼び出し時に手続きの種別に応じた処理を行うことができる.

この手法では, 手続き呼出しのたびにタグ値の判別を行うことになる. しかし, 特別なハードウェアを持たない計算機ではタグ値の判別に要するコストが大きい. そのため, 効率的に実現することは難しい.

動的コード生成　タグ判定によるオーバヘッドを削減するために, 実行時にコードを生成する手法を開発した. この手法では, 新たに手続きデータが生成される度に, 手続きの種別に応じて適切な処理を行うコードが生成される. 呼出し時には手続きの種別を判断せず, 手続き値が指す番地へ制御を移すことができる. 手続きの種別の判別に起因するオーバヘッドは, 閉包とエスケープ手続きの場合で一回のジャンプ命令の実行だけである. その他の種別の場合はオーバヘッドはない. また, 手続き生成に要するコストはタグ法と同等である.

	User	System	Total
Scheme thread	0.01	0.02	0.03
C Threads (kernel)	0.03	0.23	0.26
C Threads (user)	0.10	2.47	2.57

表 1　スレッド生成時間 (単位は秒)

6　評価

Scheme Threads の効率を評価するために, スレッド生成時間とコンテキスト切替え時間を測定した. Mach3.0 上の C Threads の実行時間も併せて測定を行った. これらの測定は図 3に示す環境で行った. 実行時間の測定には `getrusage` システムコールを用い, 10ms 単位の精度で測定を行った. 結果を示す表には, ユーザプログラムの実行時間を User の欄に, カーネルの実行時間を System の欄に, そして両者の和を Total の欄に示している.

6.1　スレッド生成時間の測定結果

`thread-fork` を 1,000 回実行するのに要する時間を測定した. 結果を表 1に示す. C Threads では, Mach カーネルが提供するスレッド間で切替えが行われる場合 (kernel) と, コルーチンによって実現されたスレッド間で切替えが行われる場合 (user) の両方の結果を示している.

ハードウェア	Luna88K(CPU: MC88100 25MHz, 1CPU, MM: 32MB)
OS	Mach3.0 (MK78 + UX38) (シングルユーザモード)
C 言語処理系	cc -O
Scheme 言語処理系	コンパイラが生成するコードを想定し, ハンドコーディング

図 3　評価に用いたシステムの構成

	User	System	Total
Scheme thread	1.30	0.98	2.28
C Threads (kernel)	0.36	4.46	4.82
C Threads (user)	1.20	0.00	1.20

表 2　コンテキスト切替え時間 (単位は秒)

このように, Scheme Threads は C Threads と比較してスレッド生成の効率が高いと言える. コルーチンによって実現されたスレッドの生成時間が Mach カーネルのスレッドの生成時間に比べて著しく遅くなる原因はスタック領域を確保する際のオーバヘッドによるものと考えられるが, 正確な分析はしていない.

6.2　コンテキスト切替え時間の測定結果

`thread-yield` を 100,000 回実行するのに要する時間を測定した. 結果を表 2に示す. C Threads では, Mach カーネルが提供するスレッド間で切替えが行われる場合 (kernel) と, コルーチンによって実現されたスレッド間で切替えが行われる場合 (user) の両方の結果を示している.

Scheme Threads とコルーチンによって実現された C Threads を比較すると, ユーザ空間のプログラムを実行している時間は同等である. カーネルの処理を行っている時間は Scheme Threads の方が大きいが, Scheme Threads のベンチマークプログラムはコンテキスト切替えの際にシステムコールを発行しない. そのため, これはメモリ参照に伴うページ割付けのオーバヘッドによるものと推測される.

7　関連する研究

Scheme 言語に並行性を導入する研究がいくつか行われている. そのような研究に, MultiScheme[8], PaiLisp[6] がある.

MultiScheme では, 並行に実行するプログラムを `future` 構文の中に記述する. `future` 構文の値を参照しようとしたスレッドは, このプログラムの評価結果が得られるまで停止させられる. このように, MultiScheme は Scheme Threads にくらべて高水準の基本操作を提供していると言える. しかし, `future` は付録 Aに示すように Scheme Threads の基本操作を用いて容易に実現することができる. また, Scheme Threads では親子関係を含む任意の方法でスレッドを関連付けることができるという点で, MultiScheme と比較して柔軟性が高い.

PaiLisp は `future` 構文を含む様々な構文を備えている. また, `call/cc` を拡張することによって, 継続をプロセスの制御に用いることを可能としている. この機能も Scheme Threads の基本手続きを用いて容易に実現可能である. Scheme Threads による PaiLisp の `call/cc` 手続きの実現を付録 Bに示す.

これらの他に, 並行性を備えた Scheme 言語上で継続の概念を拡張する研究として [5] がある.

8　まとめ

Scheme 言語のためのユーザレベルスレッドライブラリ Scheme Threads の実現法, および Scheme Threads の動作に適した処理系の実現法について述べた. Scheme Threads は様々な並行基本操作を実現する際に用いることができるような基本的な手続きを備えている.

二つのベンチマークの結果から, Scheme Threads が Mach3.0 のカーネルが提供するスレッド機能を上回る性能を持っていることが分かった. また, コルーチンによって実現された C Threads と比較しても, 大きな差がないことが分かった. 特に, スレッド生成時間は C Threads の 2 つの実現と比較すると大変優れていると言える.

謝辞

Mach3.0 について様々な点で協力して下さったオムロン株式会社の Mach 研究グループの皆様に感謝します.

参考文献

[1] William D. Clinger and Eric M. Ost, Implementation Strategies for Continuations, *Proceedings of the 1988 LISP and Functional Programming Conference*, pages 124–131, 1988.

[2] William Clinger and Jonathan Rees (Editors), Revised^4 Report on the Algorithmic Language Scheme, LISP Pointers, Volume IV, Number 3, July-September 1991.

[3] Eric C. Cooper and Richard P. Draves, C Threads, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, May 1990.

[4] Eric C. Cooper and J. Gregory Morrisett, Adding Threads to Standard ML, Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[5] R. Hieb and R.K. Dybvig, Continuations and Concurrency, *1990 ACM Conf. on the Priciples and Practice of Parallel Programming(PPoPP)*, March 1990.

[6] Takayasu Ito and Tomohiro Seino, On PaiLisp Continuation and its Implementation, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 73–90, June 1992.

[7] M. Katz and D. Weise, Continuing Into the Future: On the Interaction of Futures and First-Class Continuations, *1990 ACM Conf. on Lisp and Functional Programming*, Nice, France, June 1990.

[8] James S. Miller, MultiScheme: a Parallel Processing System Based on MIT Scheme, MIT/LCS/TR-402, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.

[9] Norman Ramsey, Concurrent Programming in ML, Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, April 1990.

[10] John H. Reppy, First-Class Synchronous Operations in Standard ML, Technical Report TR-89-1068, Department of Computer Science, Cornell University, December 1989.

[11] John H. Reppy, Asynchronous Signals in Standard ML, Technical Report TR-90-1144, Department of Computer Science, Cornell University, August 1990.

[12] Mitchell Wand, Continuation-Based Multiprocessing, *Proceedings of the 1980 LISP Conference*, pages 19–28, 1980.

付録 A　Scheme Threads による future の実現

Scheme Threads による future の実現を以下に示す．future の意味は Katz と Weise[7] に従っている．

```
(define (future body)
  (let* ((mutex (mutex-alloc)) ; 以下の 3 つの変数を操作するための mutex
         (cvar (condition-alloc mutex)) ; future の評価が終了したことを知らせる条件変数
         (resolved #f) ; body はすでに評価ずみか
         (value #f) ) ; 評価結果

    (define (touch)
      (mutex-lock mutex)
      (if (not resolveed) (condition-wait cvar))
      (mutex-unlock mutex)
      value )

    (call-with-current-continuation
     (lambda (k)
       (thread-fork k touch 0)
       (let ((r (body)))
         (mutex-lock mutex)
         (cond (resolved (mutex-unlock mutex)
                         (k (lambda () r)) )
               (else (set! resolved #t)
                     (set! value r)
                     (condition-broadcast cvar)
                     (mutex-unlock mutex)
                     (thread-exit #f) )))))))
```

付録 B　PaiLisp の call/cc の Scheme Threads による実現

```
(define (pailisp-call/cc proc)
  (let ((k-thread (thread-self)))
    (call-with-current-continuation
     (lambda (k)
       (proc (lambda (v)
               (cond ((eq? thread-self k-thread) (k v))
                     (else (thread-suspend k-thread)
                           (thread-set-continuation! k-thread
                                                     (lambda (d) (k v)))
                           (thread-resume k-thread) ))))))))
```