入れ子関数を利用したマルチスレッドの実現

田畑悠介[†] 八杉昌宏[†,††]
小宮常康[†] 湯淺太一[†]

プログラム中に記述されたスレッドをすべてOSが提供するスレッドに対応させると大きなオーバヘッドが発生するため，ユーザレベルのマルチスレッドを使用することが有効である．本論文ではユーザレベルのスレッド，より正確には，言語処理系により実現される言語レベルのマルチスレッドの実現の方法として入れ子関数を用いる方法を提案する．入れ子関数は，定義されたときの環境でlexicalスコープの変数にアクセスすることができ，そのポインタは一種のクロージャとして利用することができる．各関数に自分と同等の計算を続けるための入れ子関数を持たせ，その入れ子関数のポインタを保存してスレッドの未処理の計算（継続）を先行して実行したい場合に呼び出せるようにすることによってマルチスレッドを実現する．また，これをGCC（GNU C Compiler）の入れ子関数をそのまま用いて実現するとオーバヘッドが無視できないので，それを改善する方法についても述べる．

Implementation of Multiple Threads by Using Nested Functions

Yusuke Tabata,[†] Masahiro Yasugi,[†,††] Tsuneyasu Komiya[†]
and Taiichi Yuasa[†]

Using OS-level threads for all threads described in a program incurs a large overhead. Thus it is better to use user-level threads. In this paper, we propose a method to implement user-level multiple threads, more precisely, high-level language threads realized by a language system by using nested functions. A nested function can access the lexically scoped variables in the definition-time environment and its pointer can be used as a kind of closure. To implement multiple threads, every function has its own nested function to continue its equivalent computation and save the pointer of the nested function to be called later to early execute the thread's unprocessed computation (continuation). Since the naive implementation using GNU C Compiler's nested functions incurs a considerable overhead, we also discuss an improvement to enhance performance.

1. はじめに

マルチスレッドを利用すると複数のプロセッサを持つ計算機の能力を有効に利用することのほかに，分散環境などにおいてデータのアクセスの時間がかかるときに別の処理を行うように処理の順序を変更することによって遅延隠蔽が容易に行えるようになる．プログラムの記述のしやすさという面でも，独立した処理の流れをそれぞれスレッドとしてプログラム中に明示して記述することによって動作を分かりやすく記述することができる．また，関数型言語のような言語では処理の順序を規定せずに記述を行うため分離した処理の流れをスレッドに1対1で対応させると仕様の理解が容易になる．

しかし，マルチスレッドを用いる際，これをユーザレベルではなくOSレベルのスレッドに対応させたりすると，スレッドの生成，破棄，切替えなどのオーバヘッドが大きなものとなるため，プログラムを記述するときにスレッド操作のオーバヘッドを意識することが必要になり，プログラマの自由度が損われてしまう．たとえば，関数型言語の実行にマルチスレッドを用いると細粒度のスレッドが多く発生するため，OSレベルのスレッドなどに対応させた場合，大きなオーバヘッドが発生する．

ここで，十分低いオーバヘッドでプログラミング言語におけるマルチスレッドが実現できるのであれば，計算に含まれる並列性をマルチスレッドで表現することによって高い実行効率とプログラムの記述のしや

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communication and Computer Engineering, Graduate School of Informatics, Kyoto University
†† 科学技術振興事業団さきがけ研究21,「情報と知」領域
"Information and Human Activity", PRESTO, Japan Science and Technology Corporation（JST）

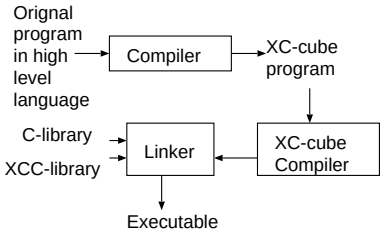


図 1 XC-cube を用いた処理系
Fig. 1 Compilation using XC-cube.

すさを両立させることができる．本論文では，GCC（Gnu C Compiler）[1]が C 言語への拡張として実現している入れ子関数を用いて細粒度のユーザレベルのマルチスレッド，より正確には，言語処理系により実現される（高水準）言語レベルのマルチスレッドを低いオーバヘッドで実現する方法を提案する．

また，我々は XC-cube という C 言語をベースとした並列処理向けの実装用言語の設計を行っている．実装用言語とその処理系を用いて高水準言語の実装を行うと，低レベルの最適化やレジスタ割当て，アセンブリコードの生成などの処理を実装用の言語に担当させることができるので処理系の実装が容易になる．

XC-cube を利用した高水準言語の処理系は，**図 1** のように実装しようとする高水準言語のプログラムから XC-cube のプログラムへの変換を行うコンパイラと XC-cube のコンパイラを組み合わせることによって実現することができる．本論文で提案する方式は，このような処理系を実現する際に高水準言語のプログラムに含まれるスレッド操作を C 言語や XC-cube などの実装用言語のプログラムに変換を行うのに用いることができる．

本論文中ではマルチスレッドの記述ができる高水準言語の例として Scheme [2]風の言語を用いて説明を行う．ただし，今回は，XC-cube ではなく GNU C Compiler の拡張機能とコンパイラを用いている．また，GCC の入れ子関数を用いたマルチスレッドの実現ではオーバヘッドが無視できないので，XC-cube では性能の改善を目指している．

本論文では以下に 2 章でマルチスレッドに関する各種の研究について議論し，3 章では入れ子関数の仕様や実装について説明した後に，4 章で提案する方式について説明を行う．そして提案する方式について 5 章で議論を，6 章で評価を行い，最後に 7 章でまとめを行う．

2. 関 連 研 究

言語レベルのマルチスレッドを持つ高水準言語の多くの実装ではスレッドの実行の状態を管理するためにスタックを利用しており，スタック中には実行途中の関数フレームが保存されている．スタックを用いない方法としては，関数フレームをヒープ上に確保する方法が考えられるが，この場合ヒープ上からの関数フレームのための領域の確保や解放の効率がスタックを用いる場合と比べて低くなる．できるだけスタック上から関数フレームを確保できるようにする方法は，スタックを実際に複数用意する方法と，単一のスタックを巧妙に利用する方法がある．

実際にスタックを複数用意する方法はスレッドライブラリの提供する C 言語レベルのスレッドなどによって使用されている方法である．これは既存のプログラムおよびコンパイラをほとんどそのまま使用することができるなどの長所がある．その一方で，このようなスレッドを用いると OS の機能を呼び出す場合には，そのオーバヘッドがあるほかに，仮想アドレス空間の無駄な消費という問題もあり，言語レベルで発生するさまざまな粒度の多数のスレッドを実現するためには不適切である．

それに対して単一のスタックを用いて，言語処理系で高水準言語のマルチスレッドを実現する方法としては OPA [3]や StackThreads [4]などで実現されている方法がある．これらの方法ではスレッドが（はじめて）サスペンドする際にスタック中のサスペンドしようとするスレッドの関数フレームの内容をヒープに退避する．スレッドがサスペンドしない限りはスタックを用いた高速なフレームアロケーションが可能であり，さらに StackThreads では新しいスレッドをできる限りそのままスタック上で実行するため，スレッドの生成・消滅のコストはきわめて小さい．また OPA の実装では，2 つのバージョンのコードを生成して，サスペンドが発生しない限りは高速なコードで実行し，関数フレームをヒープに退避したスレッドの実行を継続する際には別のコードを実行することにより，C 言語の機能のみを用いてスタックとヒープの双方に関数フレームを確保することが可能になっている[5]．

このような方式を採用した場合には関数フレームをスタックからヒープに保存するときやスタックに復帰するときにオーバヘッドが発生する．また，これらの方式では関数フレームがヒープやスタックの別の場所に移動することがあるため，関数フレーム中の変数のアドレスを他の関数に渡すことができないという問題がある．

StackThreads を発展させた StackThreads/MP [6]では，関数フレームを指すフレームポインタと，ス

タックトップを指すスタックポインタを独立させ，関数フレームを移動させずに，別スレッドの関数の実行に切り替えることを可能とし，C 言語レベルのマルチスレッドがマルチプロセッサ上でも実現されている．その実装では C コンパイラの出力アセンブリコードの後処理を行っている．

また，Cilk [7] は OPA と同様に 2 つのバージョンのコードを生成するが，OPA とは異なり，高速版のコードも関数フレームをヒープ上に確保する．Cilk 処理系は Cilk で記述されたプログラムを C のプログラムに変換することによって実現されている．

3. 入れ子関数の仕様と実装

この章では入れ子関数の仕様と一般的な C 言語の実装におけるスタックの使い方について説明したのち，GCC の入れ子関数の実現法について説明する．

3.1 入れ子関数の仕様

本来の C 言語の仕様に Pascal のような入れ子関数は存在しないが，GCC は拡張機能としてこの機能を提供している．入れ子関数とは**図 2** の関数`f_in`のように関数の中で定義される関数である，GCC ではローカル変数の宣言が可能な場所での定義が可能であり，その名前はローカル変数と同様に定義されるブロック内で有効である．入れ子関数は定義が行われた時点の環境に従って外側の関数の仮引数やブロックのローカル変数やラベル，他の入れ子関数にアクセスすることもできる．つまり入れ子関数へのポインタは，入れ子関数本体だけでなく，その定義時の環境を合わせて組としたクロージャ[8]（closure）として利用できる．C 言語の場合は入れ子関数のポインタを取得して関数呼び出しの引数としたりグローバル変数に代入したりするなどの手段によって入れ子関数を定義した関数（環境）の外で入れ子関数を呼び出すことができる．図 2 の例では関数 `f` の呼び出し中に定義された入れ子関数 `f_in` へのポインタを関数 `g` の呼び出しの引数としており，`g` では仮引数の関数ポインタ`fn` を通じて入れ子関数`f_in` の呼び出しが行われている．

取り出された関数ポインタを用いて呼び出された入れ子関数からは外側の関数の変数だけではなくラベルもアクセスでき，`goto` 文を使用することによって非局所脱出を行うことも可能である（図 2 の例では `f` のラベル `L` へと `f_in` から `goto` を行っている）．

また，ブロック内のローカル変数へのポインタをそのブロックの実行完了後に使用してはならないように，ブロック内の入れ子関数もブロックの実行完了後に使用してはならない．

```
int f(int a){
  __label__ L;
  int x;
  int f_in(int b){
    x = a+b;
    goto L;
    return 0;
  }

  x = g(f_in);
  f_in(5);
 L:
  return x;
}

int g(int (*fn)(int)){
  fn(10);
  return 0;
}
```

図 2 入れ子関数の例
Fig. 2 An example of nested function.

3.2 スタックを用いた実装

一般的な手続き型言語の関数呼び出しでは，関数の引数，リターンアドレス，呼び出した関数のフレームポインタ，callee save レジスタ，ローカル変数などのための領域がスタックの先端に確保される．この領域は関数フレームと呼ばれる．C 言語では `alloca` 関数の呼び出しなどによって関数の実行中に関数フレームの大きさが変化する．このような言語の実装では関数フレーム中のデータをそのポインタからのオフセットで参照するためのポインタとスタックトップを指す 2 つのポインタが使用される場合が多い．スタックトップを指すポインタはスタックポインタと呼ばれ，関数フレーム中のデータをオフセットで参照するためのポインタはフレームポインタ（もしくはベースポインタ）と呼ばれる．関数呼び出しによって新たなフレームがスタックトップに確保される．また，仮引数やローカル変数はその値がレジスタ上になければ，フレームポインタからのオフセットでアクセスされることが多い．

3.1 節で説明した入れ子関数を実現する場合には外側の関数の関数フレーム中の変数にアクセスする手段が必要になるが，実装の 1 つの方法として，1 つ外側の関数の関数フレームを指すポインタを関数フレーム中（もしくはレジスタ）に持ち，このリンクをたどることによって外側の関数フレーム中の変数へアクセスするという方法がある．このリンクはスタティックリンクと呼ばれる．

3.3 GCC の入れ子関数の実装

GCC の入れ子関数の実装では上で述べたスタティックリンクを用いている．入れ子関数を定義した関数が，入れ子関数名の有効範囲で入れ子関数を呼び出す場合には，スタティックリンクを特別な追加の引数として，入れ子関数本体の処理を行うコード（命令列）を呼び出せばよい．

関数ポインタの取得において通常の関数の関数ポイ

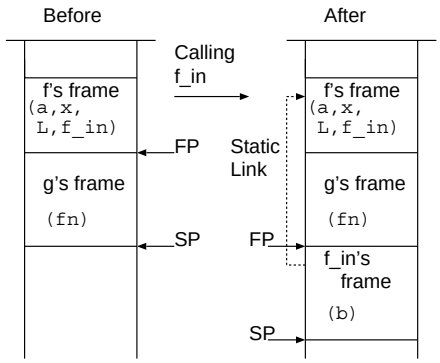


図 3 入れ子関数の実行
Fig. 3 An invocation of nested function.

ンタの実体は，その関数本体の処理を行うコードの先頭のアドレスと考えてよいが，入れ子関数の関数ポインタの実体を単なるコードのアドレスとするのは簡単ではない．入れ子関数の関数ポインタは入れ子関数本体と定義されたときの環境の組であるクロージャ（のポインタ）として利用できなくてはならず，単なるコードのアドレスでこの組を表す必要がある☆．

GCC ではこの問題をトランポリン[8]と呼ばれる機構を使うことによって解決している，トランポリンとは入れ子関数を定義した際のフレームポインタの値をスタティックリンク用のレジスタ（特別な追加の引数）にセットしてから入れ子関数本体のコードのアドレスへジャンプする数命令の短いコードのことであり，スタック上に動的に生成される☆☆．そのスタック上のトランポリンのコードのアドレスを関数ポインタとすることによって入れ子関数のポインタをクロージャとして取り出すことができ，ほとんどのアーキテクチャにおいて動作する．この手法によって GCC は入れ子関数を実現しているが，スタック上にコードを動的に生成することや，アーキテクチャによってはプロセッサの持つ命令キャッシュを明示的にフラッシュする必要があることなどのオーバヘッドが発生する．

入れ子関数のポインタを取り出して関数外に渡して図 2 のように呼び出した際のスタックの状態を**図 3** に示す．関数 `f` の中で取り出された`f_in` の関数ポインタを `g` が仮引数 `fn` として受け取って呼び出した際にはスタックのトップに関数フレームが確保されるが，スタティックリンクは入れ子関数を定義した元の関数 `f` の関数フレームを指すように設定される．

4. 提案する実現方法

我々は入れ子関数の機能を用いて，スタック中の関数フレームを移動することなしに高水準言語のマルチスレッドを実現することを提案する．入れ子関数を用いると，スタックの途中にある関数フレームのローカル変数を参照するクロージャをスタックトップで呼び出すことができるため，各関数に自分と同等の計算を続けるための入れ子関数を持たせ，その入れ子関数のポインタを保存して，別のスレッドの実行が進めなくなった際にスレッドの未処理の計算（継続）を先にスタック上で行うことができる．入れ子関数自体はすでに GCC に実装されているため，この方法は実現が容易である．この章では，マルチスレッド機能を持つ高水準言語で記述されたプログラムを入れ子関数を用いた C 言語のプログラムへ変換する提案手法について説明する．

4.1 高水準言語の仕様

提案手法はさまざまな高水準言語からのプログラムの変換を想定しているが，説明のために次のような仕様の言語を用いる．

- シンタックスは Scheme のものを用いるが，型付きで C 言語に近い仕様とする．すべての処理は式として記述し，文に相当するものは`void` 型とする．
- ポインタ（参照），関数，配列，構造体なども C と同様とし，C のポインタに対する`&`,`*`演算子に対しては`ref`，`deref` を用いる．
- 後に説明するようなマルチスレッドに関する機能を仕様に追加している．

4.1.1 スレッドのモデル

この高水準言語のマルチスレッド機能の仕様は言語レベルのものであり，ここで決める仕様はプロセッサ数を想定しない．この高水準言語におけるスレッドは，**活性状態**（active）か**中断状態**（suspended）のいずれかの状態をとる．中断状態のスレッドの実行は中断されていて先へは進まず，活性状態のスレッドの実行は実際にプロセッサにスケジューリングされることによって進行する．スケジューリングの fairness は保証されないが，スレッドが 1 つでも活性状態であれば実行は進行する．このスレッドは次のような仕様を持つものとする．

- スレッドは実行時に活性状態で生成され，実行を開始する．

☆ 通常の関数の場合も定義時の環境でグローバル変数などにアクセスできるが，その環境はコンパイルとリンクの際にコードに埋めこまれる．

☆☆ gcc-2.95.2 ではサポートするすべてのアーキテクチャにおいて，スタティックリンクの 1 段目はつねにレジスタに確保されることになっている．

- 生成時には実行すべきコードが与えられ，実行が終了したスレッドは消滅する．
- 活性化状態のスレッドは自らの実行を**中断**（suspend）し，中断状態となることができる．その際，その継続（continuation）を保存できる
- 他のスレッドは，保存された中断状態のスレッドの継続を用いて，中断状態のスレッドを活性状態にして実行を**再開**（resume）させることができる．

4.1.2 マルチスレッドに関するプリミティブ

前項で与えたモデルのための機能を次のようなプリミティブを用いて高水準言語の仕様に追加する．

- `(thread-create` `((`⟨型式⟩$_1$ ⟨変数⟩$_1$ ⟨式⟩$_1$`)` $\cdots$ `(`⟨型式⟩$_n$ ⟨変数⟩$_n$ ⟨式⟩$_n$ `))` ⟨式⟩$'_1$ $\cdots$ ⟨式⟩$'_m$`)`
  まず各⟨式⟩$_i$（$1 \le i \le n$）の値 v_i を求める．次に v_i を保持する新しい場所に⟨型式⟩$_i$ の型の⟨変数⟩$_i$ を束縛する変数束縛を追加した環境で⟨式⟩$'_j$（$1 \le j \le m$）を実行するスレッドを生成する．新しい場所は生成されたスレッドが実行を終了するまで有効である．
- `(thread-suspend (`⟨変数⟩`)` ⟨式⟩$_1$ $\cdots$ ⟨式⟩$_n$`)`
  現在実行中のスレッドの実行再開のための継続を保持する新しい場所に⟨変数⟩を束縛する変数束縛を追加した環境で⟨式⟩$_i$（$1 \le i \le n$）を実行した後に，現在実行中のスレッドを中断状態にする．⟨式⟩$_i$（$1 \le i \le n$）では，⟨変数⟩を保存できる．
- `(thread-resume` ⟨式⟩`)`
  中断状態のスレッドの継続を⟨式⟩の値にとり，そのスレッドを活性状態にして実行を再開させる．

4.2 スケジューリングの基本方針

1つのプロセッサ上で，高水準言語における活性状態のスレッドのうち同時には1つを実行するものとする．高水準言語レベルでの活性状態は実装のレベルでは**実行中状態**（running）と**実行可能状態**（runnable）に分けられる．実行中のスレッドが他の実行可能なスレッドか新たに生成され実行を開始するスレッドへ制御を移すこと，つまり実行中状態から実行可能状態へと遷移することを（実行を）**譲渡**（yield）すると呼ぶ．逆に実行可能なスレッドが制御をもらうことを（実行を）**継続**（continue）すると呼ぶ．

高水準言語における，スレッドの生成（開始），実行終了，実行中断，実行再開は実装レベルでもそのまま対応するスレッドの状態遷移となるが，スレッドの生成と実行再開では活性状態となるスレッドがすぐに実行を継続して実行中状態になるのか実行可能状態になるのかという実装上の自由がある．ここではスレッドを生成したスレッドは生成された実行を譲渡しないものとする．つまり，スレッドは実行中状態で生成され，実行可能状態で再開させられるものとする．

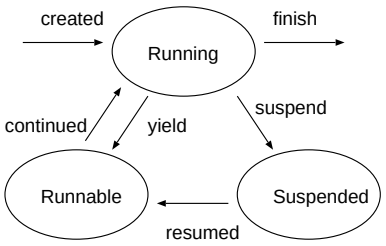


図 4 スレッドの状態
Fig. 4 Status of a thread.

これらの状態遷移を図にすると**図 4** のようになる．

4.3 スケジューリングの実装方式

提案する実装方式では，C の関数として，高水準言語の関数に対応する通常の関数，およびそれと同等な処理の継続が可能な入れ子関数，スケジューラ，および補助関数を用いる．また，データとしてスレッド管理スタックを用いる．

実装方式の基本的なアイディアは次のとおりである．

- C 言語の暗黙的継続のほかに，入れ子関数の形の明示的継続を用いる．
- スレッドが中断するなどして計算が進められなくなって他のスレッドに制御を移したい場合には，スケジューラを呼び出して，スレッド管理スタックから他の実行可能なスレッド（の明示的継続）を選択し，その実行を入れ子関数の呼び出しによって継続させることができる．

4.3.1 各関数とデータの機能と構成

この項では提案方式による変換で出力される C 言語のプログラム中の各要素について説明する．

- 通常の関数
  高水準言語レベルの関数に対応する C 言語の関数である．自分と同等の計算を続けるための入れ子関数を持ち，入れ子関数に切り替わっても実行が継続できるように計算がどこまで進んだかを覚えている．スレッド生成に関しては，新しいスレッド用コードをそのまま持ち，新しいスレッド用コードに対応する入れ子関数も持つ．
- 入れ子関数
  高水準言語レベルの関数に対応しており，通常の関数と同等の計算を表す．スケジューラから呼び出されることで，入れ子関数の途中から計算を続けることができる．高水準言語レベルの返り値の授受のためにも利用される．高水準言語レベルの関数ではなく新しいスレッド用コードに対応する

入れ子関数もある．スレッド生成に関しては，通常の関数と同様に，新しいスレッド用コードとそれに対応する入れ子関数を持つ（入れ子関数も入れ子関数を持つ）．

- スケジューラ
  スレッド管理スタックを調べて，実行可能なスレッドを選び，入れ子関数を呼び出すことでそのスレッドの実行を継続させる．ただし，スケジューラは間接的には再帰的に呼び出されるので，スケジューラ呼び出し間の関数フレームがすべて使用済みの場合は，スタックを縮めるために，外側のスケジューラへと非局所脱出する．
- 補助関数
  スレッドの再開に関して補助関数を用意している．
- スレッド管理スタック
  提案方式は実行中でないスレッドの継続となる入れ子関数の関数ポインタを継続の状態とともにCのスタックとは別のスタックを用いて管理する．スタック中の各エントリの状態には次のものがある．
  - `new_runnable`：スレッドはrunnableで，明示的継続も暗黙的継続も有効である．
  - `new_suspended`：スレッドはsuspended状態で，明示的継続も暗黙的継続も有効である．
  - `runnable`：スレッドはrunnableで，明示的継続のみ有効である．
  - `suspended`：スレッドはsuspended状態で，明示的継続のみ有効である．
  - `scheduled`：そのエントリで管理していた明示的継続はすでに使用されており，有効な継続ではない．

4.3.2 スケジューリングの方式

スレッドはrunning，runnableおよびsuspendedの3つの状態をとるが，C言語による実装レベルでは，**図5**のように，8つの状態に分けて，その間を状態遷移すると考えると分かりやすい．図5の上半分では，実行が通常の関数または新しいスレッド用コードで行われており，下半分では，対応する入れ子関数内で行われている．また，runnableについては，新しいスレッドを生成して実行を譲渡したためにrunnableとなっているのか，実行を中断後，再開させられたためにrunnableとなっているのかを分けて考えるとよい．runnable状態のうち，上の3つは明示的な継続と暗黙的な継続のどちらも有効である．明示的な継続となる入れ子関数の呼び出しは図5の破線の遷移に相当する．

以降，この項では，高水準言語での関数呼び出し・リターン，スレッドの生成・終了，スレッドの中断・再開に対する具体的なスケジューリングの方式について述べる．また，スケジューラの動作について具体的に述べる．

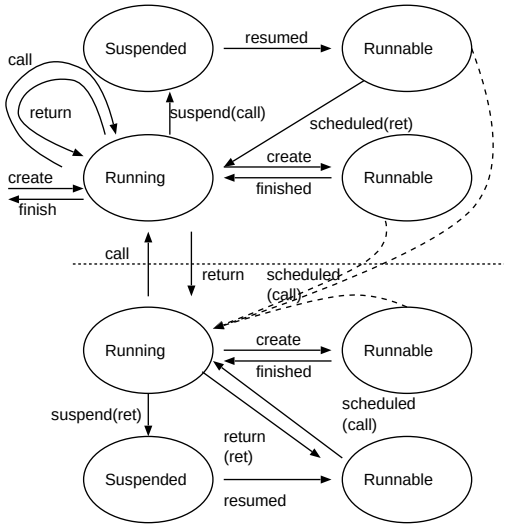


図5 スレッドの状態
Fig. 5 Status of a thread.

- 関数の呼び出し・リターン
  高水準言語の関数呼び出しの際は，通常のCの関数を呼び出す．このとき，継続となる入れ子関数のポインタ(*)を引数として渡しておく．高水準言語の呼び出された関数の処理は，呼び出された通常のCの関数で続けられるが，最初に自分と同等の処理を行う入れ子関数を定義する．
  高水準言語のリターンの際は，それが通常のC言語の関数がそのまま終了する形であれば，単純にC言語の呼び出し元にリターンする．その場合，高水準言語のリターン後の処理は，C言語の呼び出し元で続けられる．
  一方，それまでに入れ子関数による実行に切り替わっていて入れ子関数が終了する形であれば，スケジューラへとリターンすることになるので，C言語でのリターンの前に，自スレッドの状態がrunnable状態となるようにスレッド管理スタックのエントリをrunnableというフラグ付きの継続で更新する．そのためのエントリの位置は，スケジューラが入れ子関数を呼び出す際の引数で渡しておく．また，エントリに保存される継続としての入れ子関数のポインタには，上記(*)のポインタを用いる．その場合，高水準言語のリターン後の処理は，スケジューラがそのエントリに保存さ

れている runnable なスレッドを選んでその入れ子関数を呼び出すことで，継続される．

- スレッドの生成・終了
  高水準言語のスレッドの生成の際には，C 言語では新しいスレッド用のコードの実行にそのまま移ることにするが，その前に，新しいスレッドに渡す引数の評価とその値を保存する場所の確保をし，また，生成元スレッドの実行を先行して継続できるように，生成元スレッドの継続（入れ子関数のポインタ）を`new_runnable`というフラグでスレッド管理スタックに push する．新しいスレッドの処理は，新しいスレッド用のコードで続けられるが，最初に自分と同等の計算を行う入れ子関数を定義する．
  高水準言語のスレッドの生成後の生成元スレッドの処理は，新しいスレッド用のコードが最後に呼び出したスケジューラからのリターンにより継続されるか☆，スレッド管理スタックに push した`new_runnable`な継続をスケジューラが選んでその入れ子関数を呼び出すことで，継続される．
  高水準言語でのスレッドの終了の際は，それが新しいスレッド用のコードがそのまま終了する形であれば，スケジューラを呼び出す☆☆．
  一方，高水準言語でのスレッドの終了の際に，それが新しいスレッド用コードに対応する入れ子関数が終了する形であれば，そのままスケジューラへとリターンすればよい．
- スレッドの中断・再開
  高水準言語のスレッドの中断の際には，それが通常の関数または新しいスレッド用コードによる実行中であれば，`new_suspended`というフラグ付きの継続をスレッド管理スタックに push してから，スケジューラを呼び出す．スレッドが再開させられた後の処理は，スケジューラからのリターンによりそのまま継続されるか，中断時にスレッド管理スタックに push された後にフラグが`new_runnable`に変わった継続をスケジューラが選んでその入れ子関数を呼び出すことで，継続される．
  一方，高水準言語のスレッドの中断の際に，それが通常の関数または新しいスレッド用コードの，入れ子関数による実行中であれば，スケジューラから直接呼び出されているので，スケジューラへとリターンすればよい．ただし，リターンの前に，自スレッドの状態が suspended 状態となるようにスレッド管理スタックのエントリを suspended というフラグ付きの継続で更新する．そのためのエントリの位置は，スケジューラが入れ子関数を呼び出す際の引数で渡しておく．スレッドが再開させられた後の処理は，中断時に更新したスレッド管理スタックのエントリのフラグが`runnable`に変わった継続をスケジューラが選んでその入れ子関数を呼び出すことで，継続される．
  スレッドを再開させるには，スレッド管理スタック中のスレッドの継続のフラグが`new_suspended`であれば`new_runnable`，`suspended`であれば`runnable`にそれぞれ変更する．
- スケジューラ
  スケジューラは，スレッド管理スタックを調べて，実行可能なスレッドを選び，入れ子関数を呼び出すことでそのスレッドの実行を継続させる．入れ子関数からのリターン後，基本的には，この動作を繰り返す．ただし，繰返しのたびに，スケジューラを呼び出して中断したスレッドの継続のフラグが`new_suspended`から，`new_runnable`に変わっていないかチェックし，変わっていた場合には，スケジューラから単純にリターンすることでそのスレッドの実行を継続させる．
  また，スケジューラは間接的には再帰的に呼び出されるので，スケジューラ呼び出し間の関数フレームがすべて使用済みの場合は，スタックを縮めるために，外側のスケジューラへと非局所脱出する．そのためには，1 つ外側のスケジューラが呼び出されたときのスレッド管理スタックのトップの位置を覚えておき，古いトップと現在のスレッド管理スタックのトップの間のエントリの状態がすべて`scheduled`になっていることを確認すればよい．
  スケジューラの具体的なコードは付録の関数`scheduling`を参照してほしい．

4.4 変換手法

ここでは，具体的なプログラム例を用いて，提案する変換方式について説明する．

4.4.1 関数の呼び出しとリターン

関数の呼び出しを行う**図 6** は**図 7** のコードに変換される．高水準言語の関数の呼び出しは C 言語の関数呼び出しに変換され，返り値の受け渡しも通常のコードでは C 言語の返り値の受け渡しに変換される．実行

☆ 実際には後述するように，スケジューラを呼び出さないで済ませる場合について高速化が行われる．

☆☆ 実際には後述するように，生成元スレッドの明示的な継続が`new_runnable`であれば，スケジューラを呼び出しても，すぐにリターンできるので，スケジューラを呼び出さないで済ませる場合について高速化を行う．その際は`new_runnable`な明示的な継続をスレッド管理スタックから pop して捨てる．

```
(define (int (f (int n)))
  (h (g n)))
```

図 6 関数呼び出しとリターン（元のコード）
Fig. 6 Function call and return (original code).

```
int f(cont c_p, int n){
  int ln = 0;
  int t1, t2;
  int tmp1, tmp2;
  void *f_c(thst_ptr cp, reason rsn){
    switch(rsn){
    case rsn_cont:
      switch(ln){
      case 1: goto L1; case 2: goto L2;
      } return;
    case rsn_retval:
      switch(ln){
      case 1: return (void *)&t1;
      case 2: return (void *)&t2;
      } return;
    } return;
    ln = 1;
    t1 = g(f_c, n);
  L1:
    tmp1 = t1;
    ln = 2;
    t2 = h(f_c, tmp1);
  L2:
    tmp2 = t2;
    *(int *)(c_p(cp, rsn_retval)) = tmp2;
    cp->c = c_p; cp->stat = thr_runnable;
    return;
  }
  ln = 1;
  tmp1 = g(f_c, n);
  ln = 2;
  tmp2 = h(f_c, tmp1);
  return tmp2;
}
```

図 7 関数呼び出しとリターン（変換後のコード）
Fig. 7 Function call and return (translated code).

```
(thread-create ((int i (+ x 2)) (int j (* y 2)))
   (g (* i i)) (h (+ j j)))
```

図 8 スレッドの生成と終了（元のコード）
Fig. 8 Thread creation and termination (original code).

```
  v1 = x + 2;
  v2 = y * 2;
  ln = label-num; thst_top->c = nested-func;
  thst_top->stat = thr_new_runnable;
  thst_top++;
  {
    int i = v1; int j = v2;
    int ln = 0;
    void *nthr_c(thst_ptr cp, reason rsn){
      switch(rsn){
      case rsn_cont:
        switch(ln){
        case 1: goto L1; case 2: goto L2;
        } return;
      case rsn_retval:
        switch(ln){} return;
      } return;
      ln = 1;
      g(nthr_c, i * i);
    L1:
      ln = 2;
      h(nthr_c, j + j);
    L2:
      return;
    }
    ln = 1;
    g(nthr_c, i * i);
    ln = 2;
    h(nthr_c, j + j);
  }
  if((thst_top-1)->stat != thr_new_runnable)
    scheduling();
  else thst_top--;
```

図 9 スレッドの生成と終了（変換後のコード）
Fig. 9 Thread creation and termination (translated code).

がどこまで進行したのかを記録する擬似的なプログラムカウンタとしてローカル変数`ln`を用意し，関数呼び出しなどの実行を譲渡する可能性のある場所で更新を行っている．関数呼び出しの際には入れ子関数のポインタを渡すことによって，呼び出された関数はそのポインタを継続として利用できる．この継続が呼び出された際には，入れ子関数の最初の`switch`文によって適切な場所から実行が継続される．

実行が入れ子関数内で行われる場合には，リターンは入れ子関数の引数として与えられたスレッド管理スタックのエントリに外側の関数の引数として与えられた呼び出し元の関数の継続をセットして，スケジューラにリターンする．この場合返り値を直接渡すことができないため，入れ子関数に呼び出した理由を表す引数を追加し，それを用いて返り値をセットすべき変数のアドレスを取得して返り値を渡す．

4.4.2 スレッドの生成と終了

スレッドの生成を行う**図 8** の`thread-create`は**図 9** のコードに変換される．スレッド生成後の生成元スレッドの`new_runnable`な継続をスレッド管理スタックにpushし，そのまま生成されたスレッドの処理を行う．生成されたスレッドのコードにも，実行を途中から継続するための入れ子関数を持たせる．

入れ子関数でスレッドが終了した際にはそのままスケジューラにリターンし，通常のコードでスレッドが終了した際にはスケジューラを呼び出せばよい．ただし，実際には，スケジューラを呼び出す代わりに，継続の状態が`new_runnable`であれば，明示的な継続をpopして，そのまま実行を継続する高速化を行っている．

また，図 9 には示していないが，入れ子関数用の変換後のコードについては，どこから継続するかを示すラベルを図 9 の最後（高速化されたスケジューラの呼び出しの後）に追加する必要がある．

4.4.3 スレッドの中断と再開

スレッドの中断を行う**図 10** は，入れ子関数用には

```
(thread-suspend (cc) (set! x cc))
```

図 10 スレッドの中断（元のコード）
Fig. 10 Thread suspention (original code).

```
{
  thst_ptr cc = cp;
  cc->c = nested-func;
  cc->stat = thr_suspended;
  x = cc;
  ln = label-num;
  return;
}
Llabel-num:
```

図 11 スレッドの中断（変換後の入れ子関数用コード）
Fig. 11 Thread suspention (translated code for nested function).

```
{
  thst_ptr cc = thst_top++;
  cc->c = nested-func;
  cc->stat = thr_new_suspended;
  x = cc;
  ln = label-num;
  scheduling();
}
```

図 12 スレッドの中断（変換後の通常のコード）
Fig. 12 Thread suspention (translated normal code).

図 11，通常のコード用には**図 12** のコードに変換される．

スレッドの中断は，スレッドの実行が入れ子関数で行われていれば，スレッド管理スタックのエントリを suspended の継続で更新してから，そのエントリの位置を再開用のデータとして保存するコードを実行後，スケジューラへとリターンする（図 11）．この場合，再開させられた後の処理については，明示的な継続となる入れ子関数の呼び出しによって図 11 中のラベルの位置から継続される．

一方，スレッドの実行が通常のコードで行われていれば，new_suspended の継続をスレッド管理スタックに push してから，そのエントリの位置を再開用のデータとして保存するコードを実行後，スケジューラを呼び出す（図 12）．この場合，再開させられた後の処理については，明示的な継続となる入れ子関数の呼び出しによって図 11 中のラベルの位置から継続されるか，入れ子関数が呼び出されていなければ，スケジューラからのリターンによって暗黙的な継続を用いて継続される．

高水準言語の thread-resume は，C 言語の補助関数thr_resume の呼び出しへ変換される．thr_resume の具体的なコードは付録を参照してほしい．

```
(define (void (f))
  (let ((cont a) (cont b))
    (thread-create ()
       (thread-suspend (c) (set! b c))
       (while 1 (f2 (ref a) (ref b))))
    (while 1 (f2 (ref b) (ref a)))))

(define (void (f2 ((ref cont) ar)
                  ((ref cont) br)))
   (thread-resume (deref ar))
   (thread-suspend (c) (set! (deref br) c)))
```

図 13 スタックが縮小できなくなる例
Fig. 13 Example of unshrinkable stack.

5. 議　　論

5.1 非局所脱出の安全性

非局所脱出では C のスタック上の脱出先のスケジューラのフレームよりトップ側に使用中の関数フレームが存在してはならない．これを保証するためには，スレッド管理スタックのエントリがすべて scheduled になっていれば十分である．なぜなら C スタックにおいて，使用中のフレームがあれば，スケジューラが呼び出されたときの管理用スタックのスタックトップよりもトップ側に scheduled でないエントリが来るということが不変条件として成立しているからである．

ただし，先に述べた変換手法では，高水準言語レベルの関数リターンのために，継続する入れ子関数を直前に使用していたエントリに設定するとき，使用してもかまわないエントリよりもトップ側を使用することがある．また，現在はスケジューラ間での非局所脱出だけを行っているが，入れ子関数から入れ子関数を定義した関数への非局所脱出も利用するようにすると，スタック領域を回収する機会が増えると考えられる．

5.2 スタックのメモリ消費

提案する方式では，スレッドの実行継続は（子スレッド終了後にそのまま親スレッドが継続できる場合を除くと）スケジューラからのリターンもしくはスケジューラからの入れ子関数の呼び出しによって行われ，スタック上にネストしたスケジューラの間の関数フレームはスケジューラに制御が戻った際にスケジューラの非局所脱出によって回収される．

このため**図 13** のように 2 つのスレッドが相互にブロックしつつ動き続けるような場合などに，入れ子関数から通常の関数が呼び出され，そこからさらにスケジューラが呼び出されることが繰り返されると，その間のスタックが回収できなくなりスタックを消費し続けてしまうという問題がある（**図 14** では最初のスレッド A が f で生成したスレッド B が中断し，スケジューラがスレッド A を継続させるために f の入れ子関数を

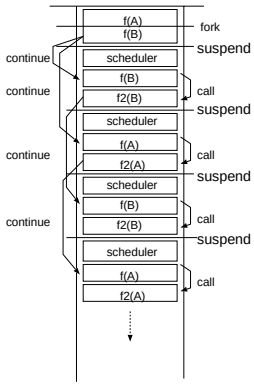


図 14 スタックが縮小できない例
Fig. 14 Unshrinkable stack.

呼び出し f の入れ子関数は f2 を呼び出す．f2 が中断してスケジューラを呼び出すと生成されたスレッドの継続のための関数を呼び出す．この繰返しによってスタックは単調に伸びてしまう）．

これを解決するためには関数フレームをヒープに確保するコードも生成して適当なタイミングで実行をそのコードに切り替えることが考えられるが，どのようなタイミングで切り替えるべきかという点で，さらなる検討が必要である．

5.3 入れ子関数のオーバヘッド

ここまで説明した GCC の入れ子関数をそのまま用いる方式では，トランポリンのコードの生成や命令キャッシュのフラッシュのためのオーバヘッドが発生する．これを解決するためには，フレームポインタとコードのアドレスを取り出すことのできる，関数ポインタとは互換性のないポインタを用いることが考えられる．開発中の XC-cube では，新しい仕様（構文，型など）の入れ子関数を追加して，軽量クロージャとして利用できるようにすることを検討している．

また，GCC の入れ子関数をそのまま用いると，外側の関数のローカル変数は入れ子関数からもアクセスされるため，関数呼び出し時には呼び出し元で必ずスタック上に退避されることになり，通常のコードの効率が低下する．入れ子関数が存在しないときと同様にローカル変数をレジスタに割り当てた状態で実行を行うには，入れ子関数が呼び出される前に外側の関数のローカル変数の内容をスタック上に復元することが必要である．ここで，callee save レジスタについては StackThreads/MP [4] で行われているようにスタックトップから各関数フレーム中に存在する callee save レジスタを復帰するとよい．

このような方法を用いると通常のコードの実行を損なわない入れ子関数を実現できることが期待できる．

5.4 入れ子関数の用途

入れ子関数はマルチスレッドの実現以外の他の用途にも使用可能である．たとえば，遅延タスク生成[9]を実現しようとする際に仕事の分割を行うためのコードを入れ子関数内に記述し，他のプロセッサが仕事を要求しているときにその関数の仕事を分割して割り振るというようなことや，GC の際にスタック上に存在するポインタのスキャンを行うといったことが可能である．

5.5 他の方式との比較

提案するマルチスレッド実現法は関数フレームの移動を行わない StackThreads/MP のフレームポインタとスタックポインタの分離を，入れ子関数からの元の関数フレームの変数へのアクセスという形で行うと考えることができる．ほかに StackThreads/MP と異なる点としては，提案方式ではスレッド生成時にスレッドの実行がスレッドを生成した関数内で開始されるため，それぞれのスレッドでローカル変数などを共有する仕様であっても容易に実現できるという点，提案方式は GCC の入れ子関数を用いることによって容易に実装できるという点があげられる．

Cilk [7] は，本論文で考えているような遅延隠蔽などのためのスレッドの切替えではなく，タスクスチールに基づく負荷分散のためにスレッドを利用しているので，本方式との単純な比較はできない．ただ，Cilk では，提案方式や StackThreads/MP と比べると関数フレームをヒープ上にも確保している分，効率が低下する．

StackThreads/MP などの他の方式ではスレッドの制御の多くはライブラリとして実現されており，それらを使う際には関数呼び出しが必要になる．それに対して，提案する方式はプログラムの変換を行う処理系を前提としてるため，スレッドの制御の多くが変数への値の代入などで行うことができ，より高速なスレッド操作を実現できる可能性がある．

6. 評 価

6.1 通常プログラムの実行効率

提案方式はスレッドの実行を継続するための入れ子関数をすべての関数の中に生成する．また，StackThreads/MP はコード生成の際にアセンブリコードの後処理を行って関数の実行継続時にレジスタの内容の復帰をするためのコードを追加する．これらのオーバヘッドを測定するためにフィボナッチ関数の実行の時間を測定した．

表 1 フィボナッチ関数の実行時間（秒）
Table 1 Result of fibonacci function (seconds).

	Sparc	PentiumIII
C 言語による実行	0.24	0.25
入れ子関数を追加した場合	1.55	0.40
トランポリンの除去	0.35	0.27
StackThreads/MP	0.34	0.25

測定には StackThreads/MP が対応している gcc-2.8.1 を用い，コンパイルオプションには `-O2` を設定した．測定には Sun Blade 100（UltraSPARC-IIe 500 MHz 1 プロセッサ，256 MB Main Memory，256 KB L2 Cache）と IBM IntelliStation（PentiumIII Xeon 550 MHz 2 プロセッサ，384 MB Main Memory，512 KB L2 Cache）を用いて実行時間を測定した（**表 1**）．

StackThreads/MP は Sparc においても PentiumIII においても低いオーバヘッドで動作していることが分かる．入れ子関数を追加すると高いオーバヘッドが発生しているが，トランポリン生成のコードを除去すると，そのオーバヘッドはかなり低くなる．

6.2 スレッド生成のオーバヘッド

次にスレッド生成のオーバヘッドを測定するために，フィボナッチ関数の再帰呼び出しの片方の呼び出しに対してスレッド生成を行うプログラムを提案方式および StackThreads/MP で実行して結果を測定した（**表 2**）．ここまで説明したプリミティブを用いて記述すると**図 15** のようなプログラムになる．このプログラムはロックを使用していないが，今回は単一プロセッサ上での実験なので対応は省略した．

また，提案方式によるプログラムに対して，アセンブリコードを変更することによりトランポリンの生成を行わないコードを作成して同様の計測を行った．POSIX スレッドを用いたバージョンについても測定を行ったが，スレッドがカーネルスレッドに 1 対 1 で対応する実装しか持たない Linux（kernel-2.2.16，glibc-2.1.3）では生成されるスレッドの数が多すぎたため実行することができなかった．

提案方式が StackThreads/MP より高速に動作しているのは，6.1 節と違い，StackThreads/MP が複数のプロセッサに対応するためにポーリングを行っているためであると思われる．

6.3 スレッドの中断と再開のオーバヘッド

次に付録のプログラムを用いてスレッドの中断と再開にかかる時間を測定した（**表 3**）．測定の条件など 6.1 節と同様である．

提案方式の方が高速に動作するのは，スレッドに対する操作を StackThreads/MP が関数呼び出しとして実現しているのに対して提案方式では，関数内でのスレッド管理スタックへの操作として実現されているという点．および，StackThreads/MP ではスレッドの継続時にスタックをスキャンしてスタック上の関数フレームに保存された callee save レジスタの値の復元を行っているということが理由として考えられる．

表 2 フィボナッチ関数の実行時間（秒）
Table 2 Result of Fibonacci function (seconds).

	Sparc	PentiumIII
C 言語による実行	0.24	0.25
本研究のスレッド	1.54	0.70
トランポリンの除去	0.56	0.37
StackThreads/MP	3.6	2.1
POSIX スレッド（参考）	28.2	-

```
(define (int (pfib (int n)))
  (if (<= n 2) 1
      (let ((int x) (int y)
                    (int nn 0) (cont c 0))
        (thread-create
         ()
         (set! x (pfib (- n 1)))
         (begin
           (set! nn (+ nn 1))
           (if (<= nn 0)
               (thread-resume c))))
        (set! y (pfib (- n 2)))
        (begin
          (set! nn (- nn 1))
          (if (< nn 0)
              (thread-suspend (c0)
                              (set! c c0))))
        (+ x y))))
```

図 15 フィボナッチ関数
Fig. 15 Fibonacci function.

表 3 スレッドの中断と再開の時間（秒）
Table 3 Thread suspenstion and resumption (micro seconds).

	Sparc	PentiumIII
本研究のスレッド	10.5	4.5
StackThreads/MP	60	34

7. おわりに

本論文では入れ子関数を用いて単一の OS のスレッド上でユーザレベル，より正確には，言語処理系により実現されるマルチスレッドを実現する方法について提案を行い，多数のスレッド操作を行うプログラムでも効率良く動作することを示した．提案方式の利点としては GCC をそのまま用いることができるため実装が容易であることがあげられる．

また今後は，性能の改善のために，GCC の入れ子関数の実装および仕様を改良していくとともに，入れ子

関数の他の利用方法についても研究を進めていきたい.

参 考 文 献

1) Stallman, R.M.: *Using and Porting GNU Compiler Collection*, Free Software Foundation, Inc., for gcc-2.95 edition (1999).
2) Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol.33, No.9, pp.26–76 (1998).
3) 八杉昌宏, 龍 和夫：並列処理のためのオブジェクト指向言語 OPA の設計とその実装, 情報処理学会研究報告, Vol.96, No.82, pp.157–162 (1996).
4) Taura, K. and Yonezawa, A.: Fine-grain multithreading with minimal compiler support—A cost effective approach to implementing efficient multithreading languages, *Proc. Conferenece on Programming Language Design and Implementation* (*PLDI*), pp.320–333 (1997).
5) 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法, 情報処理学会論文誌：プログラミング, 採録予定.
6) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.60–71 (1999).
7) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (*PLDI*), Montreal, Canada, pp.212–223 (1998).
8) Breuel, T.M.: Lexical Closure for C++, *Usenix Proceedings, C++ Conference* (1998).
9) Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).

付 録

A.1 変換の例（元のプログラム）

```
(defstruct sync
  (int n)
  (cont c))

(define
  (void (f-notify ((ref (struct sync)) s)))
  (set! (fref (deref s) n)
        (+ (fref (deref s) n) 1))
  (if (<= (fref (deref s) n) 0)
      (thread-resume (fref (deref s) c))))

(define (void (f-wait ((ref (struct sync)) s)))
  (set! (fref (deref s) n)
        (- (fref (deref s) n) 1))
  (if (< (fref (deref s) n) 0)
      (thread-suspend
       (c0) (set! (fref (deref s) c) c0))))

(define (int (g (int x) (int y))) (/ (+ x y) 2))

(define (void (fw1 (int n1) (int n2)
                   ((array int) tab1)
                   ((array int) tab2)
                   ((ref (struct sync)) s1)
                   ((ref (struct sync)) s2)))
  (let ((int i n1))
    (while (< i n2)
      (set! (aref tab2 i)
            (g (aref tab1 i)
               (aref tab1 (- i 1))))
      (set! i (+ i 1))))
  (f-notify s1)
  (f-wait s2)
  (let ((int i n1))
    (while (< i n2)
      (set! (aref tab1 i)
            (g (aref tab1 (+ i 1))
               (aref tab1 i)))
      (set! i (+ i 1))))
  (f-notify s1)
  (f-wait s2))

(define (void (divide-work (int n) (int m)))
  (let (((array int) tab1 (make-array int n))
        ((array int) tab2 (make-array int n))
        ((struct sync) s1
                       (make-struct
                        sync :n 0 :cont 0))
        ((struct sync) s2
                       (make-struct
                        sync :n 0 :cont 0))
        (int i 0))
    (while (< i m)
      (thread-create
       () (fw1 1 (- (/ n 2) 1)
                 tab1 tab2 (ref s1) (ref s2)))
      (fw1 (/ n 2) (- n 2)
           tab1 tab2 (ref s2) (ref s1))
      (set! i (+ i 1)))))
```

A.2 変換の例（変換後のプログラム）

```
#include <stdio.h>

struct _thstelm;

/* 継続用入れ子関数の呼び出し理由 */
typedef enum { rsn_cont, rsn_retval } reason;

/* 継続用入れ子関数のポインタ */
typedef void *(*cont)(struct _thstelm *,
                      reason);

/* スレッド管理用スタックの要素 */
typedef struct _thstelm {
  /* スレッドの明示的継続 */
  cont c;
  /* スレッドと継続の状態 */
  enum {
    /* 停止中で, 暗黙的継続も有効 */
    thr_new_suspended,
    /* 実行可能で 暗黙的継続も有効 */
```

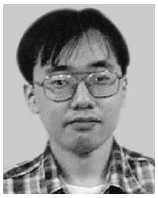
```
    thr_new_runnable,
    /* 停止中で，明示的継続のみ有効 */
    thr_suspended,
    /* 実行可能で，明示的継続のみ有効 */
    thr_runnable,
    /* 明示的継続 (cont c) は無効 */
    thr_scheduled
  } stat;
} thstelm, *thst_ptr;

thstelm thst[65536];  /* スレッド管理用スタック */

/* スレッド管理用スタックのトップ */
thst_ptr thst_top = thst;

typedef void (*schdexit)();
/* スケジューラの非局所脱出先 */
schdexit cur_schd_exit = 0;
thst_ptr cur_schd_thst_top = thst;

void scheduling(){
  __label__ L0;
  /* 元のスケジューラの情報 */
  schdexit prev_exit = cur_schd_exit;
  thst_ptr prev_thst_top = cur_schd_thst_top;
  /* このスケジューラの情報 */
  thst_ptr mythst_top = thst_top;
  void nonlocalexit(){ goto L0; }
 L0:
  cur_schd_exit = nonlocalexit;
  cur_schd_thst_top = thst_top = mythst_top;
  for(;;){
    { /* ここで元のスケジューラへの
         非局所脱出を試みる */
      thst_ptr cp;
      for(cp = prev_thst_top;
          cp < mythst_top ; cp++)
        if(cp->stat != thr_scheduled) break;
      /* 間がすべて thr_scheduled なら */
      if(cp == mythst_top) if(prev_exit)
        prev_exit();
    }
    /* runnable なスレッドを探す */
    {
      thst_ptr cp;
      cont cc;
      for(cp = thst_top-1 ; cp >= thst; cp--)
        if(cp->stat == thr_runnable
           || cp->stat == thr_new_runnable)
          break;
      if(cp < thst){
        /* 見つからなかったときは他のプロセッサからの
要求を処理すべし */
        /* 今回は何もしない */
        fprintf(stderr, "No active thread!\n");
        exit(1);
      }
 /* cp に runnable なスレッド，その継続を呼び出す */
      do{
        cc = cp->c;
        cp->c = 0; cp->stat = thr_scheduled;
        cc(cp, rsn_cont);
      }while(cp->stat == thr_runnable);
    }
    /* 直下が new_runnable なら，pop し，
       そちらに制御を移す */
    if(thst_top > thst &&
       (thst_top-1)->stat == thr_new_runnable){
      thst_top--; break;
    }
  }
  /* 元のスケジューラの情報を戻す */
  cur_schd_exit = prev_exit;
  cur_schd_thst_top = prev_thst_top;
}

/* thread-resume */
void thr_resume(thst_ptr cp){
  if(cp->stat == thr_suspended)
    cp->stat = thr_runnable;
  else if(cp->stat == thr_new_suspended)
    cp->stat = thr_new_runnable;
}

struct sync {
  int n;
  thst_ptr c;
};

void f_notify(cont c_p, struct sync *s){
  s->n++; if(s->n <= 0) thr_resume(s->c);
}

void f_wait(cont c_p, struct sync * s){
  int ln = 0;
  void *f_wait_c(thst_ptr cp, reason rsn){
    switch(rsn){
    case rsn_cont:
      switch(ln){ case 1: goto L1; } return;
    case rsn_retval:
      switch(ln){ } return;
    } return;
    {
      s->n--;
      if(s->n < 0) {
        /* thread-suspend */
        thst_ptr c0 = cp;
        c0->c = f_wait_c;
        c0->stat = thr_suspended;
        s->c = c0;
        ln = 1;
        return;
      }
    }
  L1:
    cp->c = c_p; cp->stat = thr_runnable;
    return;
  }
  {
    s->n--;
    if(s->n < 0) {
      /* thread-suspend */
      thst_ptr c0 = thst_top++;
      c0->c = f_wait_c;
      c0->stat = thr_new_suspended;
      s->c = c0;
      ln = 1;
      scheduling();
    }
  }
}

int g(cont c_p, int x, int y){
  return (x+y)/2;
}

void
fw1(cont c_p, int n1, int n2,
    int *tab1, int *tab2,
    struct sync *s1, struct sync *s2){
  int ln = 0;
  int i = n1; int i2;
  int t1, t4;
  void *fw1_c(thst_ptr cp, reason rsn){
```

```
    switch(rsn){
    case rsn_cont:
      switch(ln){
      case 1: goto L1; case 2: goto L2;
      case 3: goto L3; case 4: goto L4;
      case 5: goto L5; case 6: goto L6;
      } return;
    case rsn_retval:
      switch(ln){
      case 1: return (void *)&t1;
      case 4: return (void *)&t4;
      } return;
    } return;
    while(i < n2) {
      ln = 1;
      t1 = g(fw1_c, tab1[i], tab1[i-1]);
    L1:
      tab2[i] = t1;
      i++;
    }
    ln = 2;
    f_notify(fw1_c, s1);
  L2:
    ln = 3;
    f_wait(fw1_c, s2);
  L3:
    i2 = n1;
    while(i2 < n2){
      ln = 4;
      t4 = g(fw1_c, tab2[i+1], tab2[i]);
    L4:
      tab1[i] = t4;
      i2++;
    }
    ln = 5;
    f_notify(fw1_c, s1);
  L5:
    ln = 6;
    f_wait(fw1_c, s2);
  L6:
    cp->c = c_p; cp->stat = thr_runnable;
    return;
  }
  while(i < n2) {
    ln = 1;
    tab2[i] = g(fw1_c, tab1[i], tab1[i-1]);
    i++;
  }
  ln = 2;
  f_notify(fw1_c, s1);
  ln = 3;
  f_wait(fw1_c, s2);
  i2 = n1;
  while(i2 < n2){
    ln = 4;
    tab1[i] = g(fw1_c, tab2[i+1], tab2[i]);
    i2++;
  }
  ln = 5;
  f_notify(fw1_c, s1);
  ln = 6;
  f_wait(fw1_c, s2);
  return;
}

void divide_work (cont c_p, int n, int m){
  int ln;
  int tab1[n];
  int tab2[n];
  struct sync s1 = {0, 0}, s2 = {0, 0};
  int i = 0;
  void *divide_work_c(thst_ptr cp, reason rsn){
```

```
  switch(rsn){
  case rsn_cont:
    switch(ln)
      { case 1: goto L1; case 2: goto L2; }
    return;
  case rsn_retval:
    switch(ln){  } return;
  } return;
  while(i<m){
    {
      /* thread-create */
      ln = 1;
      thst_top->c = divide_work_c;
      thst_top->stat = thr_new_runnable;
      thst_top++;
      {
        int ln = 0;
        void *nthr_c(thst_ptr cp, reason rsn){
          switch(rsn){
          case rsn_cont:
            /* continue */
            switch(ln){ case 1: goto L1; }
            return;
          case rsn_retval:
            /* return return value location */
            switch(ln){  } return;
          } return;
          fw1(nthr_c, 1, n/2-1,
              tab1, tab2, &s1, &s2);
        L1:
          return;
        }
        ln = 1;
        fw1(nthr_c, 1, n/2-1,
            tab1, tab2, &s1, &s2);
      }
      if((thst_top-1)->stat
         != thr_new_runnable) scheduling();
      else thst_top--;
    }
  L1:
    ln = 2;
    fw1(divide_work_c, n/2, n-2,
        tab1, tab2, &s2, &s1);
  L2:

    i++;
  }
  cp->c = c_p; cp->stat = thr_runnable;
  return;
}
while(i<m){
  /* thread-create */
  {
    ln = 1;
    thst_top->c = divide_work_c;
    thst_top->stat = thr_new_runnable;
    thst_top++;
    {
      int ln = 0;
      void *nthr_c(thst_ptr cp, reason rsn){
        switch(rsn){
        case rsn_cont:
          /* continue */
          switch(ln){ case 1: goto L1; }
          return;
        case rsn_retval:
          /* return return value location */
          switch(ln){  } return;
        } return;
        fw1(nthr_c, 1, n/2-1,
            tab1, tab2, &s1, &s2);
```

```
        L1:
          return;
        }
        ln = 1;
        fw1(nthr_c, 1, n/2-1,
            tab1, tab2, &s1, &s2);
      }
      if((thst_top-1)->stat != thr_new_runnable)
        scheduling();
      else thst_top--;
    }
    ln = 2;
    fw1(divide_work_c, n/2, n-2,
        tab1, tab2, &s2, &s1);
    i++;
  }
}
```

(平成 13 年 7 月 11 日受付)
(平成 13 年 12 月 28 日採録)



田畑　悠介

1976 年生．2000 年京都大学工学部情報学科卒業．同年より同大学大学院情報学研究科修士課程に在学中．並列処理と言語処理系に興味を持つ．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．1998 年より科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

小宮　常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

湯淺　太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著),「Scheme 入門」,「C 言語によるプログラミング入門」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．