2004－OS－96　(19)
2004／6／18

マルチスレッド実行機構を考慮したプログラム実行制御法

福冨和弘[†]　乃村能成[†††]　日下部茂[††]
谷口秀夫[†††]　雨宮真人[††]

本稿では，イベント駆動によるプリエンプトなしのマルチスレッド実行機構におけるスレッド実行処理の連続性や性質に着目し，事象待ち状態をなくすプログラム実行制御法を提案する．イベント駆動によるプリエンプトなしのマルチスレッド実行機構においては，プロセッサが次々にスレッドを実行してしまうため，オペレーティングシステムが，オペレーティングシステムの想定するスケジュール単位で実行制御を行う場合に，プロセッサのサポートする実行単位に対しての制御を行う必要がある．このとき，プログラム全体の処理の流れを考慮した実行制御を行わなければ，オペレーティングシステムのスケジュールオーバヘッドが大きくなる．

A Method of Program Execution Control for Multithread Execution Mechanism

Kazuhiro FUKUTOMI,[†] Yoshinari NOMURA,[†††]
Shigeru KUSAKABE,[††] Hideo TANIGUCHI[†††]
and Makoto AMAMIYA[††]

A large number of threads run concurrently in the multithreading environments such as the processors with thread-level parallel execution mechanism. In those environments, the execution unit which a processor supports may differ from the schedule unit which operating system manages. Especially, for the machine which has the mechanism to control parallel non-preemptive event-driven thread execution, a new operating system mechanism has to be developed to control program execution with less overhead. One of the key issues of this operating system mechanism is how to schedule the conventional threads or processes which are constructed with non-preemptive event-driven threads. In this paper, we propose a program execution controlling method, which covers the waiting for an event, based on continuity and character of program processing in multithread execution mechanism.

1. はじめに

プロセッサの動作速度やクロック数は向上しているものの，画像処理など，複雑な処理を行うアプリケーションを実行するにあたっては，更なるプロセッサの高速化が要求されている．しかし，プロセッサの動作速度やクロック数を向上する手法には限界があり，近年，プロセッサのスループット向上を狙ったSMT（Simultaneous Multi-Threading）プロセッサが提案[1]され，実用化[2]されている．SMTプロセッサでは，複数のスレッドを同時に実行することができる．このように，複数のスレッドを同時に実行させるプロセッサが，今後普及すると予想される．

また，複数スレッドの基本的な実行制御機構を提供するマルチスレッド実行機構がある．マルチスレッド実行機構を有するプロセッサのように，多数のスレッドが同時に走行する環境では，プロセッサのサポートする実行単位と，オペレーティングシステム（以降，OSと略す）の想定するスケジュール単位が異なる場合がある．特に，イベント駆動によるプリエンプトなしのマルチスレッド実行機構においては，プロセッサが次々にスレッドを実行してしまうため，OSが，OSの想定するスケジュール単位で実行制御を行う場合に，プロセッサのサポートする実行単位に対しての制御を行う必要がある．このとき，プログラム全体の処理の流れを考慮した実行制御を行わなければ，OSのスケジュールオーバヘッドが大きくなる．

そこで，本稿では，イベント駆動によるプリエンプ

† 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University
††† 岡山大学工学部
Faculty of Engineering, Okayama University

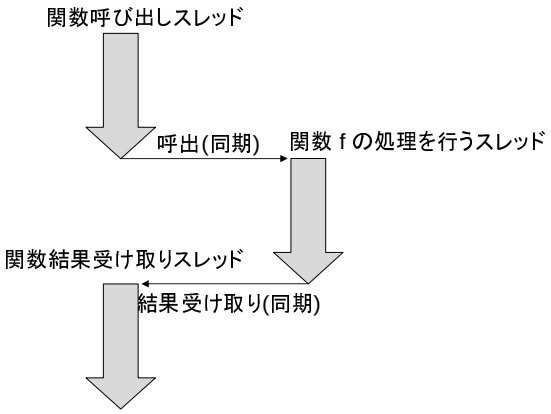

図 1 データフローモデルにおける関数呼び出し処理

トなしのマルチスレッド実行機構におけるスレッド実行処理の連続性や性質に着目し，事象待ち状態をなくすプログラム実行制御法を提案する．

以降，2 章でマルチスレッド実行機構について述べ，3 章で，マルチスレッド実行機構を考慮したプログラム実行制御法について述べる．さらに，4 章でマルチスレッド実行機構において優先度に応じた制御を行う方法について述べる．また，5 章で，ソフトウェア構成方法の具体例として，カーネルコールの実現方法について述べる．最後に，6 章でまとめる．

2. マルチスレッド実行機構

我々はマルチスレッド実行機構を有する Fuce プロセッサを開発している[3]．Fuce プロセッサがサポートするスレッドは，元来並列処理と親和性の高いデータフローモデルを基盤にしている．データフローモデルでは，たとえば，関数 f の呼び出しを行う処理では，**図 1** のように関数 f を呼び出すスレッド，関数 f の処理を行うスレッドおよび関数 f の結果を受け取るスレッドによって構成される．このように，データフローモデルでは，多くのスレッドにより構成されるため，並列処理と親和性が高い．また，継続概念に基づき，スレッドはイベント駆動により制御されるという特徴を持っている．

本章では，Fuce プロセッサがサポートするスレッドモデルと，プロセッサアーキテクチャについて述べる．

2.1 Fuce プロセッサがサポートするスレッド

Fuce プロセッサがサポートするスレッドの定義を以下に述べる．

(1) スレッドは走り切る．つまりプリエンプトされない．
(2) スレッドは走行モードを動的に変更しない．
(3) スレッドは同期カウンタ値を持つ．他のスレッドによる同期命令により，同期カウンタ値がデクリメントされることにより，実行可能状態になる．
(4) 他のスレッドとの同期は，スレッドが実行可能状態になった時点で解決されており，スレッド実行中に同期待ちを行う必要はない．
(5) スレッドの粒度は，扱うデータがレジスタに収まる程度に制限している．

以降，Fuce プロセッサがサポートするスレッドをマイクロスレッドと呼び，たとえば，Linux でいうスレッドと区別する．

2.2 Thread Activation Contoroller および Active Control Memory

Fuceプロセッサには，マイクロスレッドの生成，同期を行うための Thread Activation Controller（TAC）および Active Control Memory（ACM）機構と，複数の実行ユニットと先読みユニットがある．それらの関係を示した Fuce プロセッサの概要を**図 2** に示す．

ACM 機構は，多数のマイクロスレッドの管理情報（同期情報）を保持している．マイクロスレッドの登録命令が発行されると，TAC は ACM にマイクロスレッドの情報を登録する．同期命令が発行されると，TAC は ACM に登録されている当該マイクロスレッドの同期カウンタ値をデクリメントする．さらに，TAC は実行可能になったマイクロスレッドを実行ユニットキューにつなぐ．実行ユニットは，マイクロスレッドが実行を終えると，次のマイクロスレッドを実行ユニットキューから取り出し，実行を開始する．先読みユニットでマイクロスレッドコンテキストを先読みすることで，マイクロスレッドの実行切り替えのオーバヘッドを削減することができる．

このように，マイクロスレッド間の同期–実行処理は HW で行われる．つまり，ソフトウェアからはマイクロスレッド間の同期–実行の状態を関知できないので，この点に注意して OS を設計する必要がある．

2.3 割り込み処理および例外処理

割り込み処理は，マイクロスレッドとして扱う．具体的には，あらかじめ，割り込み処理用マイクロスレッドを準備しておき，ハードウェアが割り込みを感知すると，当該マイクロスレッドに同期を取る．これにより，割り込み処理を開始することができる．

また，マイクロスレッドはプリエンプトされないという前提があるものの，マイクロスレッド実行中に例外が起きた場合は，当該マイクロスレッドの実行を停止せざるを得ない．この場合，当該マイクロスレッドの実行を停止し，例外処理ルーチンを起動させる．例外処理終了後，例外を起こしたマイクロスレッドが継続処理可能な場合は，実行を再開する．したがって，例外処理ルーチンは実行ユニットキューにつながれることはないため，マイクロスレッドとして扱われることはない．ただし，例外処理中に，例外処理ルーチンがマイクロスレッドを生成する場合もあり得る．

ここで，割り込み処理マイクロスレッドや，例外処

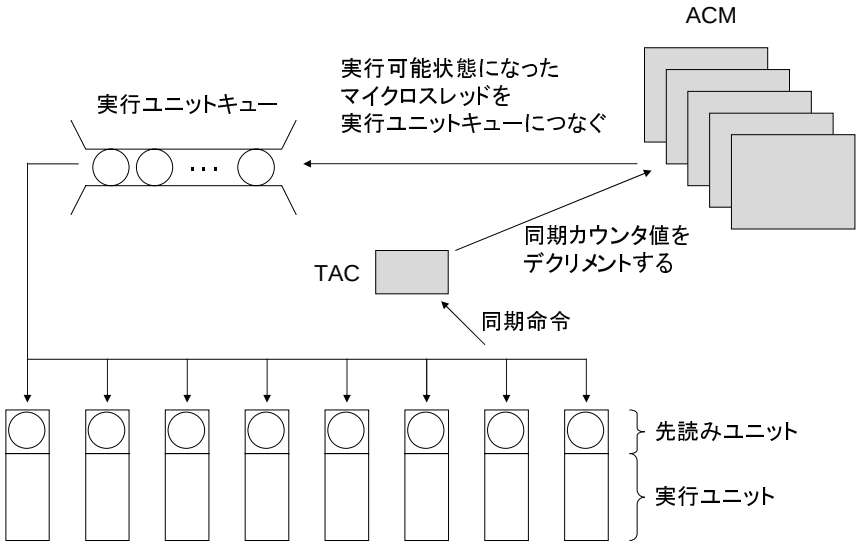

図 2 Fuce プロセッサの概要

理ルーチンが生成したマイクロスレッドは，迅速に処理されなければならない．したがって，実質上，実行ユニットキューを 2 つ用意する必要がある．

3. マルチスレッド実行機構を考慮したプログラム実行制御法

3.1 プログラム構成法

マイクロスレッドは，2.1 節に述べた定義により，UNIX 系 OS におけるスレッドに比べて，粒度が非常に小さいという特徴を持つ．また，マイクロスレッド間の同期はソフトウェアからは関知できない．そのため，Fuce プロセッサ上でプログラムの実行制御を行うには，何らかの規則にしたがって，複数のマイクロスレッドを，UNIX 系 OS におけるプロセスやスレッドに相当するものに構成する必要がある．このとき，UNIX 系 OS におけるプロセスとスレッドの関係を崩すことなく，マイクロスレッドを取り込むには，スレッドが複数のマイクロスレッドからなる**図 3** のようなモデルが望ましい．以下，プロセス，スレッドおよびマイクロスレッドの位置づけを述べる．

(1) プロセス
OS が計算機資源の割り当てに関して管理する実行単位である．1 つ以上のスレッドからなる．

(2) スレッド
OS が管理する最小の実行単位である．1 つ以上のマイクロスレッドからなる．多くの場合，多数のマイクロスレッドからなる．

(3) マイクロスレッド
HW が管理する実行単位である．同期状態などを OS から関知できない．

このようにプログラムを構成することにより，Fuce プロセッサ上でプログラムを実行することができる．

3.2 プログラム構造の比較

本節では，UNIX 系 OS のプロセス–スレッドのモデルと，Fuce プロセッサにおけるプロセス–スレッド–マイクロスレッドのモデルのプログラム構造の比較を行う．

3.2.1 ユーザスレッド

Solaris には，ユーザスレッドという概念があり[4]，OS が関与しないという点では，マイクロスレッドと共通点がある．しかし，ひとつのスレッドに属するユーザスレッドは複数のプロセッサで並列に動作できないのに対し，ひとつのスレッドに属するマイクロスレッドは複数の実行ユニットで並列に動作できるという点が異なる．

3.2.2 sleep 処理と wakeup 処理

既存の sleep 処理と wakeup 処理は，マイクロスレッド間の同期によって遮蔽される．**図 4** のような sleep 処理と wakeup 処理は，**図 5** のように，マイクロスレッド間の同期によって遮蔽される．したがって，sleep 処理や wakeup 処理を明示的に行う必要がない．これは，スレッドには明示的な SLEEP（WAIT）状態がないことを意味する．このため，UNIX 系 OS では，sleep 処理や wakeup 処理の中で，スレッド切り替えなどのスケジューリング処理を行う必要があるのに対し，提案するプログラム実行制御法では不要になる．また，既存の逐次的なプログラムをこのようなマイクロスレッドにコンパイラが分割することにより，プログラマは，基本的にデータフローモデルを気にすることなく，逐次的なプログラムを書くことができる．しかし，このような処理を行うコンパイラの設計は，一概に容易とはいえない．したがって，コンパイラを設計する労力と，データフローモデルに基づいて，プロ

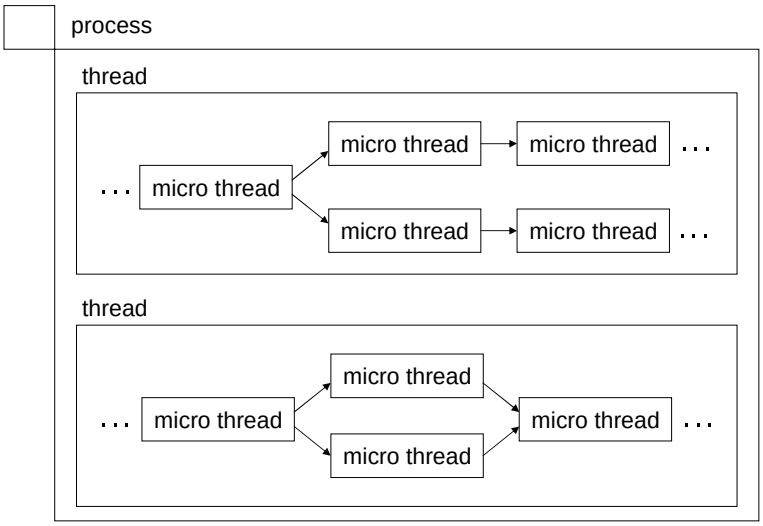

図 3 プログラム構成法

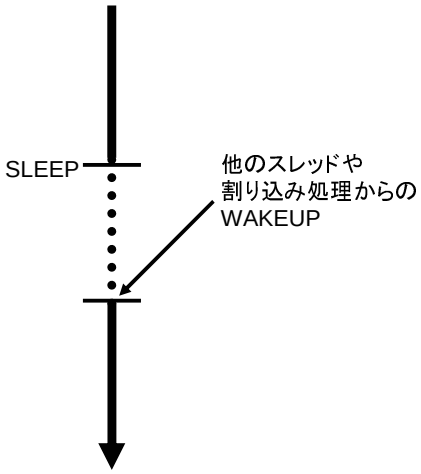

図 4 既存の sleep 処理と wakeup 処理

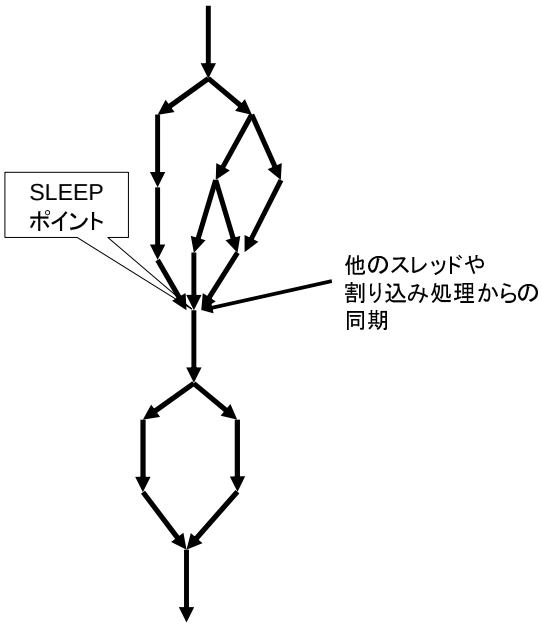

図 5 sleep 処理と wakeup 処理に相当するマイクロスレッド間の同期

グラマが SLEEP ポイント前の処理を行うスレッドと，SLEEP ポイント後の処理を行うスレッドを明示的に記述する労力とのトレードオフが肝要である．

4. 優先度に応じた制御方法について

4.1 マルチスレッド実行機構の課題

OS が優先度に基づきスケジューリングを行う対象は，スレッドである．しかし，マルチスレッド実行機構において，スレッドの実行制御を行う際には以下の課題がある．

スレッドを構成するマイクロスレッド間の同期は HW 命令で行われる．このため，OS が，マイクロスレッド間の同期状態を把握することができない．また，マイクロスレッドは走り切りであるため，マイクロスレッドの実行中にプロセッサ資源を取り上げることができない．

つまり，マイクロスレッド間の同期の渡りの部分でスレッドとしての動作を止め，他のスレッドの動作に切り替えなければならないものの，マイクロスレッド間の同期の渡りの部分を OS が把握することができないという問題がある．

4.2 スレッドの制御方法

解決策のひとつとして，同期機構を利用する以下の方法が考えられる．

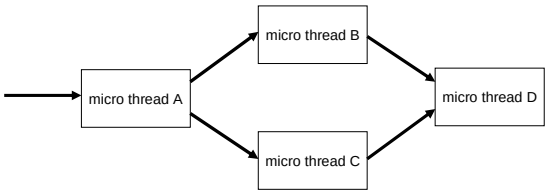

図 6 通常走行時におけるスレッドのマイクロスレッド間同期状態

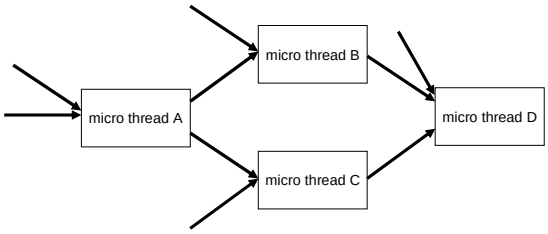

図 7 スレッドを構成するマイクロスレッドの同期カウンタ値を 1 増やした状態

動作を止めたいスレッド A があるとする．スレッド A を構成するマイクロスレッド間の様子を**図 6** に示す．スレッド A の動作を止めるには，スレッド A を構成するすべてのマイクロスレッドの同期カウンタ値を 1 増やせばよい．この様子を**図 7** に示す．これにより，マイクロスレッド間の同期が進まなくなるため，マイクロスレッド間の同期の渡りの部分でスレッドとしての動作を止めることができる．スレッド A の動作を再開したい場合は，図 6 に示すように，同期カウンタ値を 1 減らし，通常走行時の状態に戻せばよい．これにより，スレッドとしての動作を再開することができる．

5. カーネルコール実現方法

本章では，ソフトウェア構成方法の具体例として，カーネルコールの実現方法を挙げ，UNIX 系 OS におけるカーネルコール処理との比較を行う．

5.1 ソフトウェア構成方法

図 8 のように，マイクロスレッドを構成することにより，マルチスレッド実行機構上で，カーネルコールに相当する機能を実現できる．

動作の様子は，以下のとおりである．

(1) 呼び出しマイクロスレッドは，カーネルコール突入マイクロスレッドに対し，復帰先マイクロスレッドの情報を通知した上で，同期を行う．

(2) カーネルコール突入マイクロスレッドは，カーネルコール出口マイクロスレッドを生成するとともに，復帰先マイクロスレッドの情報を通知した上で，同期を行う．また，カーネルコールの種類によって，同期先のカーネルコール本処理スレッドを選択するとともに，カーネルコール本処理スレッドに対してカーネルコール出口マイクロスレッドの情報を通知する．

(3) カーネルコール本処理スレッドは，処理を終えた後，カーネルコール出口マイクロスレッドに対して同期を行う．

(4) カーネルコール出口マイクロスレッドは，復帰先マイクロスレッドに対して同期を行う．

スレッドを明示的にプログラマが記述するかについては，3.2.2 項と同様である．また，一般に，カーネルコールを行う回数は静的に決定されないので，カーネルコール処理に要するマイクロスレッドは，動的に生成する必要がある．

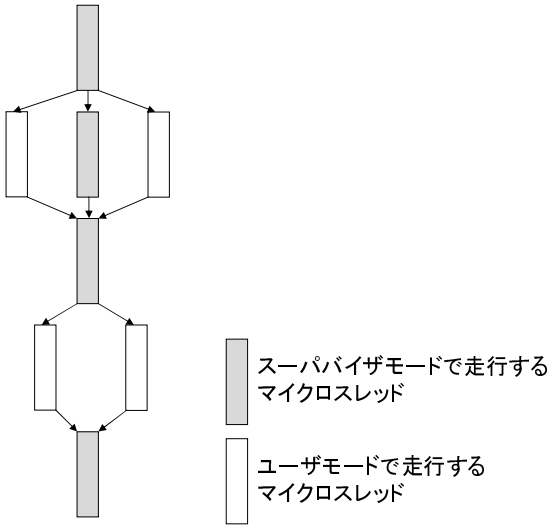

図 9 走行モードの違いによるマイクロスレッドの割り当て

5.2 既存の方式との比較

UNIX 系 OS と比較して，以下のような特徴がある．

図 9 のように，カーネル処理のうち，スーパバイザモードでの走行が必要な箇所のみ，スーパバイザモードで走行するマイクロスレッドを割り当てることができる．具体的には，たとえば，特権命令を発行する箇所をスーパバイザモードで走行するマイクロスレッドに割り当てることができる．つまり，カーネル処理すべてがスーパーバイザモードで走行する必要はなく，カーネル処理の中でも，特権命令を発行しない箇所は，ユーザモードで走ることができる．したがって，ユーザモードで走っても構わない箇所は，ユーザモードで走行するマイクロスレッドを割り当てることができる．これにより，スーパバイザモードで走行する時間を最小限に抑えることができる．たとえば，自スレッドの識別子を取得するようなカーネルコールの場合，基本的に，カーネル処理はすべてユーザモードで走ることができる．しかし，UNIX 系 OS では，カーネルコールとスーパバイザモードへの遷移が同時に行われており，このようなことを実現することはできない．

また，カーネル処理のマイクロスレッド間の同期の

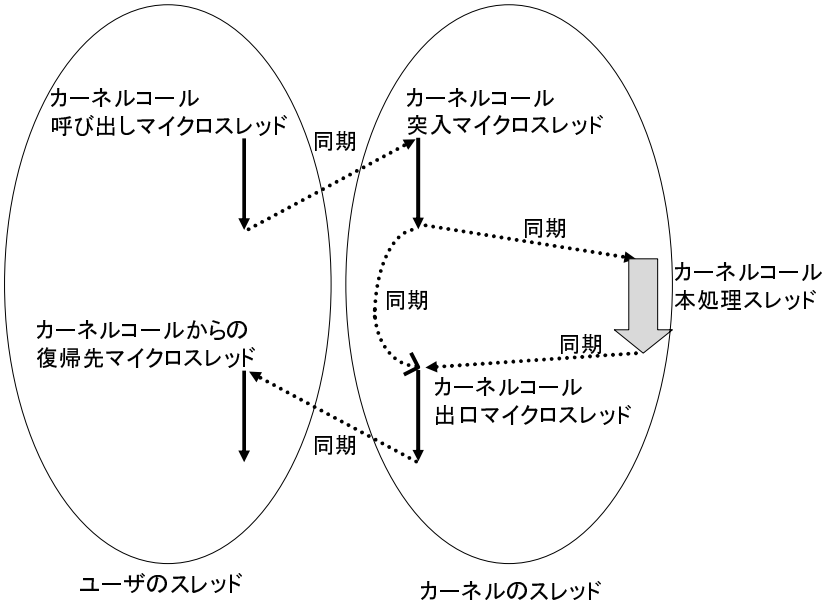

図 8 カーネルコールの実現方法

切れ目の箇所において，ハードウェアにより，別の処理に切り替わる可能性がある．したがって，カーネル内プリエンプションに相当する機能を有するといえる．

6. おわりに

本稿では，マルチスレッド実行機構について説明し，マルチスレッド実行機構を考慮したプログラム実行制御法について述べた．マルチスレッド実行機構を考慮したプログラム実行制御法では，プロセス–スレッド–マイクロスレッドの階層構造によりプログラムを構成し，OS が制御するのはプロセスとスレッドであり，マイクロスレッドについてはマルチスレッド実行機構に制御をゆだねる．また，マルチスレッド実行機構を考慮したプログラム実行制御法では，sleep 処理と wakeup 処理がマイクロスレッド間の同期処理によって遮蔽される．

また，マルチスレッド実行機構において，優先度に応じた制御を行う基本方針について述べた．スレッドを構成するマイクロスレッドの同期カウンタ値を増減させることにより，スレッドの動作を制御できる．

さらに，ソフトウェア構成方法の具体例として，カーネルコールの実現方法について述べた．マルチスレッド実行機構を考慮したプログラム実行制御法では，カーネル処理のうち，ユーザモードで処理可能な箇所はユーザモードで走行するマイクロスレッドに分割することができるので，スーパバイザモードで走行する時間を最小限に抑えることができる．また，カーネル処理が複数のマイクロスレッドにより構成されているため，カーネル内プリエンプションに相当する機能を有する．

今後は，実装と評価を行う予定である．

謝辞 本研究は，文部科学省科学研究費補助金・基盤研究（A）(2)「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号:15200002）による．

参 考 文 献

1) Lo, J.L., Eggers, S.J., Emer, J.S., Levy, H.M., Stamm, R. L. and Tullsen, D. M.: Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading, *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp. 322–354 (1997).
2) Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A. and Upton, M.: Hyper-threading technology architecture and microarchitecture: a hyperhtext history, Intel Technology J. 6,1 2002. (online journal).
3) 雨宮聡史, 松崎隆哲, 雨宮真人: 排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計, 情報処理学会研究報告, Vol. 2003, No. 119, pp. 51–56 (2003).
4) Mauro, J. and McDougall, R.: *Solaris Internals: Core Kernel Architecture*, PRENTICE HALL, INC, a Pearson Education Company (2001). (福本秀, 兵頭武文, 細川一茂, 大嶺朋之, 佐藤敬訳: Solaris インターナル: カーネル構造のすべて, ピアソン・エデュケーション (2001)).