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装と評価

笹田耕一[†] 佐藤未来子[†] 河原章二[†,☆]
加藤義人[†] 大和仁典[†]
中條拓伯[†] 並木美太郎[†]

近年，マルチスレッドプロセッサアーキテクチャの研究が盛んである．このアーキテクチャの性能を引き出すためには，システムソフトウェアのサポートが不可欠であるが，従来のモデルでは，カーネルが計算実体を管理するため，このアーキテクチャの利点を十分に活用することができなかった．そこで本研究では，マルチスレッドアーキテクチャ上で効率的に機能するユーザレベルスレッドライブラリの実現方法を検討し，実際に作成した．ライブラリは，1 チップが複数持つ実スレッドを管理し，スレッドの並列実行をサポートすることで性能を向上させる．また，プロセッサのスレッド制御命令を利用することで，高速なスレッド制御を可能にする．ユーザレベルでスレッドを管理するうえで困難な問題は OS と協調動作することで解決する．シミュレータによる評価の結果，スレッドの並列実行により最大 1.5 倍の性能向上を確認した．また，軽量なスレッド制御を実現した．

Implementation and Evaluation of a Thread Library for Multithreaded Architecture

Koichi Sasada,[†] Mikiko Sato,[†] Shoji Kawahara,[†,☆]
Norito Kato,[†] Masanori Yamato,[†] Hironori Nakajo[†]
and Mitaro Namiki[†]

Recently, there are many studies on multithreaded processor architecture. In order to get the higher performance of this architecture, support of system software is indispensable. However, from the view of performance, the advantage of the architecture has not been utilized enough without kernel supports for Architecture (Physical) Threads. In this research, we have designed and developed a thread library that works efficiently on a multithread architecture. Architecture Threads of a processor are managed on a user level library supports to parallelize threads and improve performance. Using the thread control instructions of the processor enables high-speed thread control. Inefficiency when managing threads on a user level would be improved by cooperation with OS. As a result of simulation based evaluation, up to 1.5 times higher performance has been gained by parallel execution. Moreover, we have accomplished a lightweight thread control.

1. はじめに

プロセッサアーキテクチャとして，マルチスレッドアーキテクチャが注目されている．これは，命令レベル並列性（ILP）に着目し，高性能化を図るスーパスカラアーキテクチャのアプローチでは，ILP 抽出の困難さから性能向上に限界があると考えられるためである．これに対し，マルチスレッドアーキテクチャはスレッドレベル並列性（TLP）に着目し，複数の実行命令流を 1 チップ上で同時に実行することにより性能向上を図る．その一方式として，Simultaneous Multithreading（SMT）アーキテクチャがある．これは複数の命令流を演算器などのハードウェアリソースを共有しながら並列に実行することで，実行ユニットを有効活用し，性能向上を図るアーキテクチャである．

従来のオペレーティングシステム（OS）では，CPU などの実際に計算を行う実体をカーネルが管理していた．たとえば，SMP 計算機では，CPU の個数の計算

† 東京農工大学大学院工学研究科
Graduate School of Technology, Tokyo University of Agriculture and Technology
☆ 現在，NEC シリコンシステム研究所
Presently with Silicon Systems Research Laboratories, NEC Corporation

実体をカーネルが管理し，UNIX でいうプロセスなどをそれぞれの計算実体に割り当てる．1 チップに複数の実行実体を持つマルチスレッドアーキテクチャプロセッサを従来の方法で管理しようとすると，1 チップ上に複数のプロセスが動作することになり，ワーキングセットが増大する．また，その実行単位にスレッド（または軽量プロセス）を割り当てることを考えると，その操作のためにはカーネルへ制御を移さねばならず，効率に問題がある[9]．

そこで本研究では，マルチスレッドアーキテクチャを有効に活用し高速にスレッド制御を行い，OS と連携することで効率の良いスレッド管理を可能にするユーザレベルスレッドライブラリを考案し，実際に**MULiTh**（Userlevel Thread Library for Multi-threaded architecture）というスレッドライブラリを作成した．MULiTh は，マルチスレッドアーキテクチャプロセッサ上で動作し，ユーザレベルで実行実体の管理を行い，スレッドの並列実行を実現する．スレッドライブラリのインタフェースは POSIX スレッド仕様[2] とした．また，我々が MULiTh と並行して研究開発しているオンチップマルチスレッドアーキテクチャ用 OS **Future** と協調することで，効率的なスレッドスケジューリングを実現する．マルチスレッドアーキテクチャプロセッサについても，これらシステムソフトウェアを効率良く実装できるよう設計を進めている[8]．

本論文では，2 章で対象とするプロセッサアーキテクチャについて述べる．3 章で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課題を述べる．4 章でスレッドライブラリの実現について述べ，5 章で OS との連携について述べる．6 章で評価を行う．

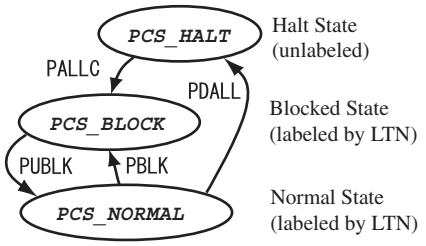


図 1 実スレッドの状態遷移
Fig. 1 Architecture thread state chart.

2. OChiMuS PE

MULiTh は，筆者らが研究，開発を進めているオンチップマルチスレッドプロセッサアーキテクチャである OChiMuS PE というプロセッサ上で実装する．本プロセッサについての詳細は文献 8) を参照されたい．

OChiMuS PE は，MIPS アーキテクチャをベースとした SMT アーキテクチャプロセッサであり，マルチスレッドをサポートするためのモジュールや命令を新たに拡張したものである．将来的には，本プロセッサを 1 つのエレメント（PE: Processing Element）とし，この PE を同一チップ上に複数搭載したオンチップマルチ SMT（OChiMuS）アーキテクチャとすることを検討している．このため，ここで示すアーキテクチャによるプロセッサを OChiMuS PE と称する．

OChiMuS PE は，1 つのプロセッサで複数の実行命令流を並列実行可能であり，1 つの命令流を処理する単位を**実スレッド**（AT: Architecture Thread）と呼ぶ．OChiMuS PE は複数の実スレッドを有している．各実スレッドは汎用レジスタなど，並列実行に必要なハードウェアリソースを個別に持つ．また，実スレッドは**図 1** で示す状態を持ち，**表 1** で示す命令によってこの状態を制御する．

1 つの実スレッドは `PALLC` 命令によって動的に**論理スレッド番号**（LTN）を設定され，1 つの論理スレッドとして割り当てられ，ソフトウェアは論理スレッド番号によって実スレッドを操作することができる．`PALLC` 以外のスレッド制御命令は，LTN で論理スレッドを指定して実行する．たとえば `PBLK` 命令は指定した論理スレッド番号を持つ実スレッドの状態を *PCS_BLOCK* にし，実スレッドの実行を一時停止する．逆に `PUBLK` 命令で *PCS_NORMAL* 状態へ復帰し，実行を再開する．また，`PDALL` 命令は実スレッドを *PCS_HALT* 状態へ戻し，解放する．`FWD` 命令はレジスタ値を転送する．これ

表 1 OChiMuS PE のスレッド制御命令
Table 1 OChiMuS PE thread control instructions.

命令	説明
`PALLC dr,sr0,sr1`	LTN を sr0，開始番地を sr1 として，論理スレッドを割り当てる
`PDALL dr,sr`	sr で指定した論理スレッドを開放する
`PBLK dr,sr`	sr で指定した論理スレッドを一時停止させる
`PUBLK dr,sr`	sr で指定した論理スレッドの一時停止を解除する
`FWD dr,sr0,sr1`	sr0 で指定した論理スレッドに sr1 のレジスタ値を転送する

* dr: destination register．制御命令の結果が返る． sr: source register

らの命令で指定した実スレッドがない場合，デスティネーションレジスタにエラーの値を書き込む．LTN は各実スレッドの専用レジスタに保持されているため，レジスタへのアクセスと同等に LTN を取得することができる．また，スレッド制御命令は 1 サイクルで実行することができる．

SMT アーキテクチャでは，ALU などの実行ユニットはプロセッサ中のすべての実スレッドで共有される．もし複数の実スレッドが同じ実行ユニットを利用しようとした場合は，競合が起こるため調停が必要となる．

SMT プロセッサとして現在製品化されている Intel 社の Xeon プロセッサ[3] では，1 プロセッサに 2 個の実スレッドを持ち，仮想的に 2CPU の SMP 計算機として動作するため，従来の SMP 向け OS を実行可能である．この方式では，1 つの実スレッドに 1 つのプロセスを割り当てられるため，1 プロセッサで扱うワーキングセットが従来に比べて広がり，キャッシュミスや TLB ミスが多発する．また，実スレッドをカーネルで管理すると，実スレッド制御を行うにはシステムコールが必要となり，効率に問題がある[9]．

これらの検討の結果，OChiMuS PE 上で動作するすべての実スレッドは 1 つのプロセスに属し，アドレス空間を共有する．そして，実スレッドの管理はユーザレベルで行うこととし，スレッド制御命令は，特権命令ではなく，ユーザレベルでも実行可能な命令とした．

3. 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課題

本章では，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課題を，従来のスレッド管理機構と比較し考察する．

3.1 スレッドの管理

従来のスレッド管理機構には，(a) カーネルレベルで実現する方法と，(b) ユーザレベルで実現する方法[7]，それらを組み合わせた (c) 2 レベルスレッド管理[11] がある．(a) は，スレッドの管理をプロセススケジューリングと同様に行うことで実現可能である．しかし，スレッド制御のオーバヘッドが大きく，スレッドスケジューラは OS で単一となり，柔軟なスケジューリングができない．(b) はスレッド制御が軽量に行え，プロセスごとに最適なスケジューラを用意することができる．しかし，単一の実行実体しか利用できないため，並列実行可能なスレッドを提供できない．(c) は (a), (b) の特徴を組み合わせ，効率の良い柔軟なスレッド管理を可能にする．しかし従来の方式では，割り当てる実行単位はカーネルが管理していたものを利用していた．OChiMuS PE ではユーザレベルでプロセッサの実スレッド制御・管理が可能であるため，これを有効に活用する方式が必要となる．

MULiTh では，ユーザレベルで動作するスレッド管理を行うため，(b) と同様の利点がある．また，実スレッドの管理も MULiTh で扱うため，並列実行可能なスレッドを実現することができる．性能向上のためには，プロセッサ上のすべての実スレッドを動作させ，処理の並列度を向上することが求められる．また，実スレッドの管理も含めたスレッドの管理コストはできるだけ低く抑える必要がある．

3.2 排他制御・同期機構

排他制御・同期機構を実装する場合，従来はスピンロックかスレッド切替えを用いて実装するのが一般的である．スピンロックではロックが解放されるとすぐにロック獲得の処理へ移ることができるという利点がある．しかし，ロック変数が格納されているメモリ参照を繰り返すため，メモリアクセスが頻発し，他の実スレッドの実行を妨げる可能性がある．スレッド切替えは，他の待ち状態のスレッドへ処理を移すことで全体の性能を向上させることができる．しかし，この方式は実行コンテキストの復帰と退避を行うため，オーバヘッドが大きい．MULiTh ではこれらの問題を回避し，効率の良い制御方式を検討する．

3.3 OS との協調

ユーザレベルでのスレッド管理には次の問題がある．

1 つは I/O に関するシステムコールやページフォールトなどでカーネル内での実行がブロックするようなケースである．理想的には，あるスレッド A がカーネル内でブロックした場合，A の代わりに待ち状態のスレッド B をレジュームしたいという場合がある．しかし，ユーザレベルでスレッドを管理しているため，カーネルはスレッド B へ切り替えることができない．

また，スレッドをプリエンプティブにすることが困難である．これは，横取りのタイミングを知ることがユーザレベルのみでは不可能であるためである．

これらの問題を解決するために，安倍らの研究[6),7] では，スレッドの横取りには SIGALRM によるシグナル通知を利用し，ブロックするような入出力は I/O プロセスを独自に生成する方式が提案されている．しかし，これらの方式では移植性は向上するがオーバヘッドが大きい．

別の解決法として，OS と協調してスケジューリングを行う研究[1),4),10] がなされている．たとえば，Scheduler Activations[1] は，カーネル内で起こった事象をユーザスケジューラに通知することで効果的なスレッ

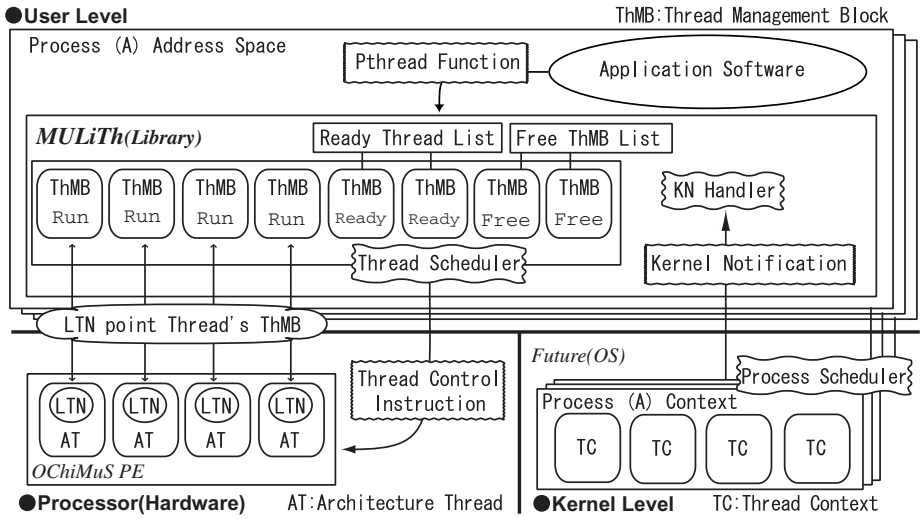


図 2 スレッドライブラリと OS，プロセッサを含めた全体像
Fig. 2 The system structure with a thread library, operating system and processor.

ドのスケジューリングを実現している．効率的なスレッドスケジューリングを行うためには，このようなカーネルとユーザレベルとの情報伝達を，できるだけ高速に行うことが要求される．

3.4 スレッドライブラリの課題

前節までで述べたように，マルチスレッドアーキテクチャに要求される点をまとめると次のようになる．

- 軽量なスレッド管理
- 新たな排他制御・同期機構
- 効率的な OS との協調，特にカーネルとユーザレベルでの情報の受け渡し

4. 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

前章で述べた考察をもとに，MULiTh をどのように設計・実装したかについて述べる．

4.1 MULiTh の概要

MULiTh はスレッド処理機構を提供するユーザレベルスレッドライブラリである．

図 2 は，提案するシステムの全体構成を示している．従来のスレッドライブラリとは違い，プロセッサ，つまり複数の実スレッドをスレッドライブラリが管理する．これにより，効率的なスレッド管理を可能にする．

スレッドは実スレッド数以上生成可能であり，スレッドライブラリはスケジューリングによりスレッドを複数の実スレッドに割り当て，それらを並列実行させる．

プログラミングインタフェースは，POSIX スレッド（Pthread）[2] の仕様に準拠したものを提供する．Pthread はスレッドライブラリの規格としては標準的なものであり，アプリケーションソフトウェアのソースレベルでの互換性を提供する．

Pthread 関数により，スレッドの生成，削除，排他制御，同期機構などを提供する．スレッドの制御は，OChiMuS PE のスレッド制御命令を利用することで，高速に行う．

また，OS と協調して動作することで，カーネル内のブロッキングの問題について回避し，プリエンプションのあるスレッド管理を実現している．これについては 5 章で述べる．

4.2 スレッド管理

各スレッドは，それぞれ**スレッド管理ブロック**（ThMB: Thread Management Block）を持つ．ThMB は，スレッドを中断したときのコンテキストの退避領域と，そのスレッドの属性を保存するための領域を持つ．現状では，ThMB はスレッドライブラリが静的に保持する．

MULiTh では，各スレッドの ThMB の先頭アドレスを，そのスレッド識別子として利用する．各スレッドがそれぞれ個別の ThMB を保持するので，その先頭アドレスも互いに違うものとなるため，識別子として利用できる．

スレッド識別子をプロセッサのスレッド制御命令に必要な LTN とする．こうすることで，ハードウェアの実スレッド管理とソフトウェアのスレッド管理を一元化することができる．また，LTN がスレッド識別子であると定義できる．LTN は，プロセッサの専用レジスタに格納されるため，レジスタアクセスと同等のコストで取得することができる．これより，自スレッ

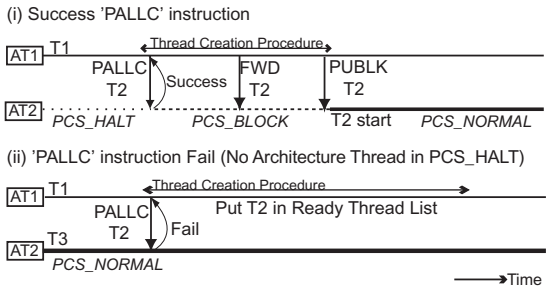


図 3 プロセッサ命令を利用したスレッドの生成
Fig. 3 Creating a new thread using Processor Instruction.

ドのスレッド識別子を取得する pthread_self 関数を軽量に実装することができる．これは，スレッド制御処理において必要なので，この取得が軽量に実装されるとスレッド制御処理全体が高速に実行できる．

実スレッドの数以上のスレッドを生成した場合，そのスレッドは待ちスレッドリストへとつなげられ，スケジューリングの対象となる．現状では，スレッドスケジューリングは FIFO として実装している．

4.3 スレッドの生成と削除

スレッド生成 スレッドを生成する処理，すなわち pthread_create 関数を実行すると，プロセッサの実スレッド割当て命令 PALLC を実行する．これが成功すれば，生成されたスレッドは実スレッドとして実行される．失敗した場合，そのスレッドを待ちスレッドリストへ格納する．

手順としては，次のようになる．

(1) 未使用の ThMB のアドレス（LTN）を 1 つ得る（失敗ならスレッド数制限による生成失敗）．
(2) PALLC 命令を実行する．
(3) (2) が成功ならば，FWD 命令で初期値の設定を行い，PUBLK 命令でそのスレッドを実行させる（生成成功）．
(4) (2) が失敗ならば，開始コンテキストを設定し，待ちスレッドリストへ格納する（生成成功）．

図 3 は，スレッド T1 がスレッド T2 を生成する様子を示している．

図 3 の (i) は生成手順の (3) の動作を示す．PALLC 命令は，状態が *PCS_HALT* の実スレッドがあれば，それに対し指定した LTN に割り当て，その実スレッドの状態を *PCS_BLOCK* にする命令である．ここでは図中 AT2 で示している実スレッドが *PCS_HALT* であるため，PALLC 命令は AT2 を対象に割当てを行う．AT2 は，LTN を T2 に設定され，*PCS_BLOCK* になる．スレッド T1 はスレッド T2 に対し，スレッド T2 実行開始時の初期設定として FWD 命令によりレジスタ値の転送を行う．その後 PUBLK 命令を発行し，スレッド T2 の実行を開始する．これにより，並列実行する実スレッドが 1 つ作られることになる．これらの制御はプロセッサ命令を利用して行うため，高速に実行することができる．

図 3 の (ii) は，(4) の動作を示しており，*PCS_HALT* 状態の実スレッドが存在しないため，PALLC 命令が失敗している．この場合，スレッド T2 の ThMB に実行開始のためのコンテキストを設定し，待ちスレッドリストへ格納する．これは，従来のユーザレベルで行うスレッド管理と同様である．PALLC 成功時に比べ，コンテキストの保存など，メモリアクセスが増える．

スレッド削除 スレッドの削除は，pthread_exit 関数が呼ばれるか，スレッドの開始ルーチンからリターンしたときに起こる☆．

削除処理は，次の順序で行う．

(1) 空き LTN リストに自スレッドの LTN を格納
(2) スレッドスケジューリング
(3) スレッド切替え

空き LTN リストに自スレッドの LTN を格納することで，その LTN および ThMB を再利用することができるようになる．次に動作させるスレッドをスケジューリングにより決定し，スレッド切替えを行う．

次に動作するスレッドが存在しなかった場合（待ちスレッドがない場合），その実スレッドを PDALL 命令により停止する．ユーザレベルでこの命令を実行し実スレッドを停止するため高速に動作する．

自発的にスレッドを切り替えたい場合，つまり pthread_yield 関数ではスレッドスケジューラを起動することにより，他の待ちスレッドへ切り替わる．

また，スレッド生成，削除に限らず，スレッド制御は各実スレッドで並列に行うことを可能にしている．このために一部のクリティカルセクションでは，MIPS プロセッサの LL，SC 命令を利用して排他制御を適宜行っている．

4.4 排他制御，同期機構

スレッドの排他制御，同期機構ではスレッドの実行をブロックする場合がある．

(1) pthread_mutex_lock 関数でロックが獲得できないとき
(2) pthread_join 関数で対象スレッドが終了することを待つとき
(3) pthread_cond_wait 関数で条件変数に対する

☆ 厳密には，デタッチされたスレッドでない場合，そのスレッドが pthread_join によって合流された時点で削除が行われる．

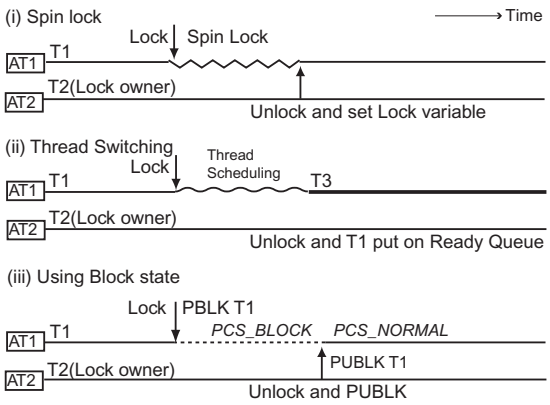


図 4 Mutex Lock の方式比較
Fig. 4 Comparison of the methods of Mutex Lock.

合図を待つとき

従来のスレッドライブラリでは，(1) はスピンロックかスレッド切替え，(2)，(3) ではスレッド切替えを用いて実装するのが一般的であるが，本研究ではスピンロックは行わず，スレッド切替えも可能ならば用いない，効率の良いブロックを実現する手法を提案する．

OChiMuS PE では，実スレッドの状態を *PCS_BLOCK*，つまり一時停止状態にする命令 PBLK，それを解除する PUBLK 命令がある．MULiTh ではこれら命令を利用して，スレッドのブロックを実現する．

図 4 では，スレッド T1 が pthread_mutex_lock 関数によってロックを獲得しようとしたが，スレッド T2 がすでにロックを獲得しているため，スレッド T1 の実行をブロックする必要がある場合の動作を示している．図 4 の (i) はスピンロックを行う様子，(ii) ではスレッド切替えによるブロックを示している．(iii) が，本研究で提案する方式である．ブロックするスレッド T1 が自分自身を対象に PBLK 命令を発行し，実スレッドの状態を *PCS_NORMAL* から *PCS_BLOCK* に遷移する．スレッド T2 がロックを解放する際，ロック解放を待っているスレッド T1 に対して PUBLK 命令を発行する．そこで，スレッド T1 は通常状態へ戻り，実行を再開する．PUBLK 命令が失敗したときには，そのスレッドはスレッド切替えによってメモリに退避されていたことが分かり，スレッド T1 を待ちスレッドリストへ登録することにより，スレッド T1 のブロックを解除することができる．

なお，MULiTh では，ブロック状態のスレッドをロック変数により管理するように実装した．あるロック変数について，どのスレッドがブロックしているかという情報は，ロック変数を参照することにより知ることができる．

本方式の利点は，スピンロックのようにループによるメモリアクセスなどを頻発せず，CPU 資源を利用しないので，他の実スレッドの動作を阻害することがない．また，PUBLK 命令 1 つを発行するだけですむため本方式でも高速にブロックからの復帰が行える．また，スレッド切替え時に発生するコンテキストの保存のようなオーバヘッドもないため，高速に動作する．

提案した方式では，スレッドのブロックが発生したときに実スレッドを必ずブロックするようにすると，すべての実スレッドがブロック状態となる危険性がある．また，この方式によるブロックは実スレッドを 1 つ占有してしまい，スレッドの並列性を損う．そのため，レジューム可能なスレッドが存在する場合はそのスレッドへ切替えを行うようにした．

しかし，ブロックした時点で待ちスレッドがなかったとしても，ブロックした後に他の実スレッドによってスレッド生成が起こった場合，このブロック状態の実スレッドには生成したスレッドを割り当てることができず☆，並列性を損なう可能性がある．

この問題点を解決するにはいくつかの方法があるが，ユーザレベルのみで対処しようとすると，大きなコストがかかることが分かった．そのため，この問題を OS との協調機構を利用して解決する．この対処については次章で述べる．

5. OS との協調

Future は，筆者らが研究を進めているマルチスレッドアーキテクチャ専用の OS であり，MULiTh と協調動作する．本章では Future の概略と MULiTh との協調について述べる．

5.1 Future におけるプロセス

Future では，プロセスは各種資源を割り当てる単位であり，プロセッサを仮想化したものと定義される．資源はたとえばアドレス空間であり，I/O 資源である．Future で仮想化するプロセッサとは，OChiMuS PE 全体であり，各実スレッドの仮想化はユーザレベルスレッドライブラリである MULiTh が行う．このプロセスが複数存在し，カーネルがこれらのプロセスを切り替えて実行する（図 2）．

Future によるプロセス切替えは次の手順で行う．すなわち，動作中の実スレッドのコンテキストの退避，保存していた実スレッドのコンテキストの復帰，である．複数存在する実スレッドはアドレス空間を共有す

☆ PALLC 命令が対象とするのは，PCS_HALT の実スレッドのみである．

るので，切替えは並列に実行される．

5.2 Kernel Notification

MULiTh は，Future と協調動作することで効率的なスレッドスケジューリングを可能にする．この協調機構を**Kernel Notification** という．これは，カーネルで起こった事象をユーザレベルへ伝える機構を提供する．

通知する事象は次のとおりである．

(1) システムコール中，カーネル内でブロックする必要がある場合
(2) ページフォールトなど，カーネル内でブロックする必要がある場合
(3) ブロックしていたスレッドが再開した場合
(4) シグナルが発生した場合

たとえば I/O に関するシステムコールを実行中，カーネル内でそのスレッドがブロックした場合を考える．カーネルはアップコールによりユーザレベルの手続きを起動する．この手続きを**Kernel Notification Handler**（KNH）という．KNH には，ブロックしたスレッドの識別子（LTN）と，KNH へジャンプした理由（この場合はシステムコール中でのブロック）などが渡される．KNH では，これらの情報をもとにスレッドスケジューリングを行う．

このとき，事象が発生する前のスレッドのコンテキストをどのように KNH に渡すかが問題となる．スケジューラアクティベーション[1] ではカーネルがその領域を用意していた．また，猪原らの研究[10] では，C-area というユーザスケジューラとカーネルの共有領域を用いて通知を最適化した．本研究では，スレッドの実行コンテキストを MULiTh が管理するそのスレッドの ThMB のコンテキスト保存領域へ直接格納する．

システムコールや例外が発生し，カーネルへ遷移する場合，カーネルはその実行コンテキストを退避する必要がある．従来の OS では，カーネルアドレス空間の領域にそのコンテキストを退避していたが，Kernel Notification ではユーザレベルの ThMB のコンテキスト退避領域にそれを格納する（**図 5**）．この方式は，実スレッドが LTN を持っており，LTN が ThMB の先頭アドレスを指すことを利用している．

従来の Scheduler Activations や C-area を利用する方式では，ThMB に相当するスレッドライブラリが管理する領域へ，ユーザスケジューラが OS から渡された実行コンテキストをコピーする必要があった．本方式では，このようなコピーが不要になるため，効率的に OS からの通知を受け取ることができる．

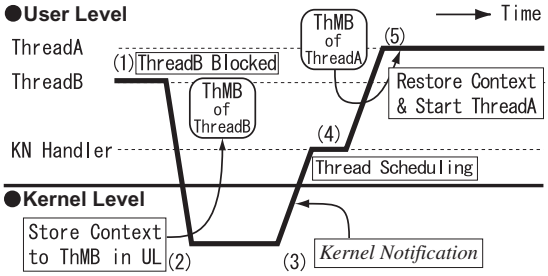


図 5 Kernel Notification の処理の流れ
Fig. 5 A flow of Kernel Notification.

5.3 プリエンプティブなスレッド切替え

OChiMuS PE では，プロセス切替えはアドレス空間の遷移と，プロセッサ内のすべての実スレッドのコンテキストの退避と復帰を保証することである．Kernel Notification によるシステムでは，コンテキストの復帰は KNH で行うことでプロセス切替えの間隔ですべての実スレッドに対し Kernel Notification が発行されることが保証される☆．

本機構によるスレッドのプリエンプションは，通常のプロセス切替えのコストに加え，KNH の処理，およびスレッドスケジューリングのオーバヘッドのみであるため，軽量に実装することができる．これは，プロセス切替えでのコンテキスト退避・復帰とスレッド切替えのコンテキストの退避・復帰を多重化するためである．

5.4 ブロック，停止している実スレッドの扱い

前節で述べたように，プロセス切替えが起こるとすべての実スレッドが KNH を実行する．これは，実スレッドの状態が *PCS_BLOCK*，*PCS_HALT* のスレッドも，同様に KNH へジャンプすることを保証する．

この機構により，プロセス切替えのタイミングでブロック状態の実スレッドに他の待ち状態のスレッドを割り当てることができ，4.4 節で述べた *PCS_BLOCK* の実スレッドが長時間残ってしまうという問題点について，少なくともプロセス切替えの間隔以上残らないように改善できる．

しかし，プロセス切替えの間隔が長い場合にはこの問題が解決できない可能性がある．これを解決するためには，システムコールで強制的にプロセス切替えを発生させるなどの方式が考えられる．

5.5 OS との競合回避

OS とスレッドライブラリが同一の ThMB を共有するため，競合が発生する可能性がある．たとえばス

☆ 競合回避中の状態を除く．

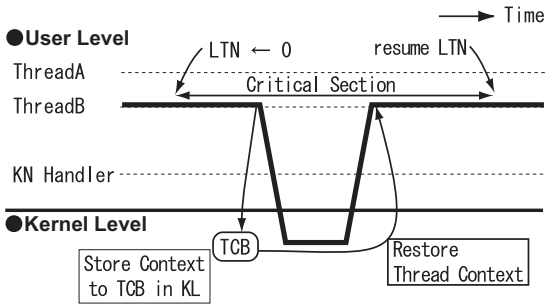


図 6 Kernel Notification の OS との競合回避
Fig. 6 Avoiding contention with OS at Kernel Notification.

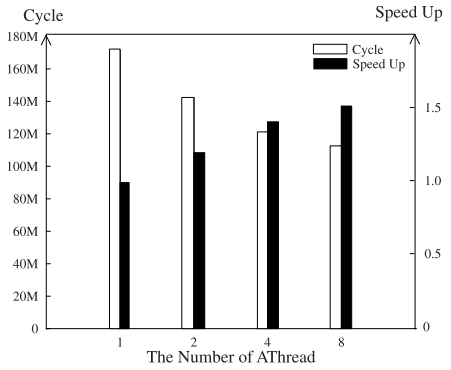


図 7 並列実行した結果
Fig. 7 A result of parallelized execution.

レッドライブラリが ThMB に対して退避を行っているとき，例外などが発生しカーネルも ThMB へ実行コンテキストを退避すると，ユーザレベルで退避中だったコンテキストは上書きされてしまう．このようなことが起こらないよう，ThMB へのアクセスの競合を回避しなければならない．

本研究では，競合回避の方法として，ユーザレベルで ThMB にアクセスする必要がある場合，実スレッドの LTN を特別な値とすることで解決した．ここでは，その特別な値を 0 とする．

スレッドライブラリが ThMB に対してアクセスするとき，実スレッドの LTN を 0 に設定する．LTN が 0 の実スレッド上でカーネルへドメイン切替えが起こった場合，コンテキストの退避先をカーネル内に用意した領域を利用する．また，復帰時は KNH へとぶことをせず，カーネル内に退避したコンテキストで復帰する（**図 6**）．

本手法により，スレッドライブラリと OS の ThMB への競合を防ぐことができる．

6. スレッドライブラリの評価

ここでは MULiTh の実行速度について述べる．

本評価は筆者らが作成している実行駆動型シミュレータ**MUTHASI**（MUltiTHreaded Architecture SImulator）[8] を用いて行った．MUTHASI は OChiMuS PE をシミュレートし，プロセッサのパラメータについて容易に設定可能となっている．また，実スレッド数を 1 に設定すると，通常の MIPS プロセッサの挙動を示す．なお，本論文ではキャッシュを機能せずに評価を行った．

プログラムの作成は binutils-2.13☆，gcc-3.2☆☆，newlib1.9.0 を用いた．MULiTh 自体の機能はスレッドの生成，削除，同期，排他制御の基本動作について実装した．スレッド制御の詳細な設定については現状ではサポートしていない．

☆ OChiMuS PE のスレッド制御命令を利用可能にしたもの．
☆☆ 最適化オプションは -O3 を設定した．

表 2 スレッド制御の性能
Table 2 The thread control performance.

評価項目	(1) 提案方式	(2) 従来方式 または (1) が利用できない場合 (*)
スレッド生成	84 (74)	135 (102)
スレッド削除	51 (42)	223 (126)
同期	202 (95)	847 (515)
	(1) 提案方式	(2) Spin Lock
排他制御 (短時間)	972	982
排他制御 (長時間)	41,461	46,656
	(1) 提案方式	(2) 従来方式
プリエンプション	523 (341)	878 (718)
OS からの通知	373 (256)	522 (418)

＊ スレッド制御命令が失敗した場合，または利用できない場合
単位は総実行サイクル数（かっこ内は実行命令数）

6.1 並列実行による高速化の評価

図 7 は，平均画素法による画像縮小プログラムを MULiTh の Pthread インタフェースを利用してマルチスレッド化したプログラムの実行結果である．実スレッドの数が 1 の場合，従来のスーパースカラアーキテクチャプロセッサ，それ以外がマルチスレッドアーキテクチャによる並列実行した結果となっている．

スレッドの並列実行によって，処理速度を向上させることができたことが確認できる．

6.2 スレッドライブラリの評価

表 2 は，スレッドライブラリの各操作の性能を測定した結果である．シミュレータの実スレッド数を 4 にして評価を行った．以下，各測定項目について述べる．

評価に利用したシグナル機構，Scheduler Activations 機構については，カーネルからの通知機構部分

のみを実装した．

- スレッドの生成
  (1) は PALLC 命令が成功した場合，(2) は従来のスレッドライブラリの動作，または PALLC 命令が失敗したときの動作である．(2) では実際に生成したスレッドが動作するには，加えてスレッド切替えのオーバヘッドがかかる．PALLC 命令が成功して新たなスレッドが実スレッドとして割り当てられたとき，処理の多くがレジスタアクセスのみですむため，高速にスレッドを生成できることが分かる．
- スレッドの削除
  (1) は，待ちスレッドがない場合の PDALL 命令によるスレッド削除，(2) は従来のスレッド削除における他のスレッドへの切替え，または MULiTh においてのスレッド切替えを示している．PDALL を実行する (1) では，スレッドの削除が 1 命令ですむので非常に軽量である．
- 同期
  (1) はプロセッサのブロック状態を利用した方式，(2) は従来のスレッド切替えによる方式の結果である．また，MULiTh でも，待ちスレッドがある場合は (2) の動作を行う．提案方式では，コンテキストの退避と復帰を行わないので，スレッド切替えを行う場合に比べて，4 倍の性能向上となった．
- 排他制御
  ロック獲得のための待ちに，(1) はプロセッサのブロック状態を利用した方式，(2) はスピンロックを利用した方式である．短時間，長時間とは，クリティカルセクションの長さである．短時間の処理では，違いはあまりみられない．しかし，長時間排他制御による保護を行うと，スピンロックの利用は性能が落ちている．これは，スピンロックによって，他の実スレッドの実行を妨げているためであり，提案する方式の有用性が確認できた．
- プリエンプション
  (1) は提案する Kernel Notification による非同期のスレッド切替え，(2) は UNIX のシグナル通知によるプリエンプションである．提案する Kernel Notification では，従来のシグナル機構における何度かのコンテキストのコピーが行われないため，プリエンプションが効率的に行われている．実際のシステムでは，プロセス切替えとスレッド切替えのコンテキストの退避・復帰を多重化するため，より高性能となる．
- OS からの通知
  (1) は提案する Kernel Notification による OS からの通知，(2) は Scheduler Activations 機構，特にそれを効率的に行う C-area [10] による通知を実装し，比較した．Kernel Notification ではスケジューラ内でのコンテキストのコピーが行われないため，高速に実行することができている．特にメモリアクセスが少なくなるため，キャッシュを有効にした際には効果があると考えられる．

7. まとめ

本論文では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について考察，実現し，それを評価した結果を示した．

評価の結果，スレッドをプロセッサが複数持つ実スレッドに割り当て，並列実行させることで性能を向上させることができることが分かった．また，プロセッサのスレッド制御命令を利用し，軽量なスレッド制御が行えることが確認できた．

本論文では OChiMuS PE と Future の利用を前提として議論を行ったが，本研究での提案はこのアーキテクチャのみに限定されるものではない．たとえば OChiMuS PE では MIPS アーキテクチャを採用していたが，実スレッドの制御を可能にするアーキテクチャであれば，ベースとなるアーキテクチャは問わない．たとえば Xeon プロセッサが実スレッド管理制御をユーザレベルで可能にすれば，本研究での提案を適用することができる．また，OS は今後 Linux などにマルチスレッドアーキテクチャ向けプロセス管理を実装し，MULiTh と協調動作の実現させる予定である．

今後の課題として，OChiMuS プロセッサの最終的な目標である，OChiMuS PE を複数オンチップに搭載する OChiMuS チップについての構想を，プロセッサ，システムソフトウェア双方の視点から考察していく．また，スレッドスケジューラについて SMT プロセッサに適した方式の研究[5] などを MULiTh に適用できないか，またそれ以外の効果的なスレッドスケジューリングの方式はないか考察したい．

本提案は，Future と連携し，実用的なアプリケーションを動作させることで，効果を発揮することが期待できる．今後，より現実的な環境による評価を通じてその可能性を明らかにしていきたい．

参考文献

1) 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 Trans. Comput. Syst.*, Vol.10, No.1, pp.53–79 (1992).
2) IEEE: *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1* (1996).
3) Intel: Intel Technology Journal (Hyper-Threading Technolog). http://www.intel.com/technology/itj/2002/volume06issue01/
4) Marsh, B.D., Scott, M.L., LeBlanc, T.J. and Markatos, E.P.: First-Class User-Level Threads, *Proc. 13th ACM Symposium on Operating Systems Principle*, Pacific Grove, CA, pp.110–121 (1991).
5) Snavely, A. and Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, *Architectural Support for Programming Languages and Operating Systems*, pp.234–244 (2000).
6) 安倍広多，松浦敏雄，安本慶一，東野輝夫：ユーザレベル軽量プロセスライブラリにおける効率の良い I/O 処理方式，情処研報 98-OS-79, Vol.98, No.71, pp.77–84 (1998).
7) 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).
8) 河原章二，佐藤未来子，並木美太郎，中條拓伯：システムソフトウェアとの協調を目指すオンチップマルチスレッドアーキテクチャの構想，コンピュータシステムシンポジウム，Vol.2002, No.18, pp.1–8 (2002).
9) 佐藤未来子，河原章二，中條拓伯，並木美太郎：SOC 時代に向けた SMT 用 OS の構想，システムソフトウェアとオペレーティング・システム，No.91-5, pp.31–38 (2002).
10) 猪原茂和，益田隆司：ユーザとカーネルの非同期的な協調機構によるスレッド切り替え動作の最適化，情報処理学会論文誌，Vol.36，No.10, pp.2498–2510 (1995).
11) 坂本　力，宮崎輝樹，桑山雅行，最所圭三，福田　晃：並列性と移植性をもつユーザレベルスレッドライブラリー PPL の設計および実装，電子情報通信学会論文誌，Vol.J80-D-I, No.9, pp.42–49 (1997).

(平成 15 年 1 月 25 日受付)
(平成 15 年 5 月 29 日採録)

笹田　耕一（学生会員）

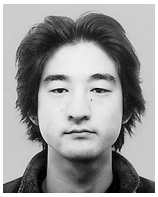
1979 年生．2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同大学工学研究科博士前期課程情報コミュニケーション工学専攻在学中．オペレーティングシステムやシステムソフトウェア，並列処理システム，言語処理系，プログラミング言語に関する研究に興味を持つ．

佐藤未来子（学生会員）

1966 年生．1990 年東京農工大学大学院工学研究科修了．同年（株）日立製作所入社，サーバシステムの設計・性能評価等に従事．2002 年より東京農工大学大学院工学研究科博士課程に在学中．オンチップマルチスレッドアーキテクチャプロセッサ，オペレーティングに関する研究に興味を持つ．

河原　章二（正会員）

1978 年生．2001 年東京農工大学工学部電子情報工学科卒業．2003 年同大学大学院工学研究科修了．同年 NEC シリコンシステム研究所に入社．プロセッサアーキテクチャ，並列処理システムに興味を持つ．



加藤　義人

2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同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在学中．プロセッサアーキテクチャに興味を持つ．

大和　仁典

2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同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在学中．オンチップマルチスレッドアーキテクチャ，メモリアーキテクチャに関する研究に興味を持つ．

中條 拓伯（正会員）

1961 年生．1985 年神戸大学工学部電気工学科卒業．1987 年同大学大学院工学研究科修了．1989 年神戸大学工学部助手を経て，現在，東京農工大学工学部情報コミュニケーション工学科助教授．1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development（CSRD）にて，Visiting Research Assistant Professor．プロセッサアーキテクチャ，並列処理，クラスタコンピューティング，高速ネットワークインタフェースに関する研究に従事．電子情報通信学会，IEEE CS 各会員．博士（工学）．

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．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等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．