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

猪原茂和† 益田隆司†

現在共有メモリ型マルチプロセッサでは，カーネルスレッドとユーザスレッドによる2レベルのスレッド管理が広く用いられている．その性能は主に，ユーザタスク内のコンテクストスイッチ，オペレーティングシステムカーネル内のコンテクストスイッチ，およびユーザタスクとカーネルの間のドメインスイッチの3種類のスレッド切り替えの頻度およびオーバヘッドによって決定される．現在のマルチタスク環境のスレッド管理機構では，理想的には必要のないスレッド切り替えが起こっており，その結果スレッド管理のコストが不必要に大きくなっている．既存のスレッド管理機構の問題点はユーザタスクとカーネルとのインタフェースにある．すなわち，(1) ユーザタスクがカーネルスレッド数を決定し，(2) カーネルがプロセッサの割り当て状況をユーザタスクに知らせず，(3) カーネルスレッドの操作がユーザタスクとカーネルとの同期的なやりとりによる，という既存のインタフェースの三つの性質が余分なスレッド切り替えの原因となっている．本論文で提案するスレッド管理機構は，共有メモリを通じた非同期的な協調動作を行うことにより，これら三つの性質を排除し，この結果マルチタスク環境において最適なスレッド切り替え動作を実現する．本論文では実装による性能測定とシミュレーションによる解析の両方で，このスレッド管理機構の有効性を検証する．

A Thread Management Mechanism for the Optimal Thread-Switching Behavior

SHIGEKAZU INOHARA† and TAKASHI MASUDA†

In shared-memory multiprocessors, thread mechanisms with the user-level and kernel-level scheduling are widely used. The performance of the thread mechanism is dominated primarily by three kinds of thread-switching overheads: context switching in user tasks, context switching in the kernel, and user/kernel domain switching. Existing thread mechanisms in a multi-tasking system performs unnecessary thread switchings because (1) user tasks determine how many threads to use, (2) the kernel does not inform user tasks of which nor how many threads are actually being assigned processors, and (3) interaction between the kernel and user tasks is synchronous. This paper presents a thread mechanism that optimizes the thread-switching overhead by avoiding these three problems; the kernel determines how many threads to use in each user task, the kernel lets user tasks know which threads are actually being assigned processors, and all interaction between the kernel and user tasks is asynchronous. Employing these three ideas, the proposed mechanism optimizes thread switchings in a multi-tasking system.

1. はじめに

スレッド管理システムは，ユーザプログラムのアドレス空間（プロセス）に仮想的なプロセッサ群（スレッド）を実現するものであり，並列アプリケーションを記述する際に広く用いられている．シングルタスクの並列マシン上では，スレッド管理システムを用いることにより，マシンの持つプロセッサ数と独立に並列プログラムを記述することができるようになる．さらにマルチタスクの並列マシン上のスレッド管理システムは，複数の並列アプリケーションがプロセッサを時分割，空間分割して共有することを可能にする．特にマルチタスク環境では，プロセッサ資源の管理のために，スレッド管理システムがオペレーティングシステムのカーネルに組み込まれる必要がある．この結果スレッド管理システムは，そのシステム上のすべてのアプリケーションから利用されることになる．

数値応用，非数値応用を問わず，スレッド管理のオーバヘッドが無視できない応用は多い．このため，これまでスレッド管理のオーバヘッドを減らすための研究が数多くなされてきた．これらの研究は主に静的解析

† 東京大学 大学院理学系研究科 情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo

手法，プロセッサ資源割り当て戦略，実行時のスレッド管理システムの三つの分野に分類することができる．静的解析手法は，ループのコンパイル手法，データフロー解析やクリティカルパス解析などにより，スレッドのオーバヘッドを静的に削減することを目的とする．プロセッサ資源割り当て戦略は，マルチタスク環境において，各アプリケーションに，いつ，どのプロセッサを，何個割り当てるかを決定する．スレッド管理システムの研究では，実行時のスレッドの動作をいかに効率的に行うかが課題となる．

本研究が対象とするのは，実行時のスレッド管理システムである．実行時のスレッド管理システムにおいては，生成，切り替え，同期，通信，消滅などのスレッドの基本操作の中でも，スレッド切り替えのオーバヘッドを削減することが特に重要である．その理由は，プロセス内，プロセス間のスレッド切り替えのオーバヘッドが，スレッド管理全体のオーバヘッドの主要な部分を占めるためである．なお仮定として，ハードウェアは共有メモリ型マルチプロセッサまたは単一プロセッサとし，オペレーティングシステムはマルチタスク，マルチユーザ，仮想記憶をもつ汎用のシステムとする．これは文献1)，2)，5)，6)，8)～11)，13)，15)，16)，21)などと同じ環境である．

本論文では，マルチタスク環境の共有メモリマルチプロセッサ上で，実行時に最適なスレッド切り替え動作を実現するスレッド管理システムを提案する．ここでいう最適なスレッド切り替え動作とは，各プロセスのスレッド操作プリミティブの列とプロセッサ割り当て戦略とがシステム実行中に与えられる場合に，これらを実現するためのスレッド切り替えの回数を最小にすることを指す．これは従来のスレッド管理システムでは実現されていなかった．

提案するスレッド管理システムでは，各プロセス内のスレッドスケジューラ（以下，ユーザスケジューラと略す）とカーネルとが，共有メモリ領域を通じて協調的，相互支援的なスレッド管理を行う．本提案のポイントは二つある．第1点は，ユーザスケジューラとカーネルとが互いに情報交換をし，相手の動作を適切に操作することを可能にした点である．第2点は，この協調動作に際してユーザスケジューラとカーネルとが同期する必要がないように，共有メモリ領域の使用法を定めた点である．これらの結果，本方式は最適なスレッド切り替え動作を実現する．また本論文では，提案方式における個々のスレッド切り替えが，従来と同等のコストで実現可能であることも示す．これは提案するスレッド管理システムが，スレッド管理全体のオーバヘッドを削減することができることを示すためである．

以下第2章で，スレッド切り替え動作の最適化という観点から従来のスレッド管理システムの問題点を明らかにする．第3章では，本論文で提案するスレッド管理システムにおけるユーザスケジューラとカーネルとの相互作用について詳しく述べる．続く第4章では，提案する方式で最適なスレッド切り替え動作が得られることを示す．第5章ではこのスレッド管理システムの有効性を，実装による性能測定とシミュレーションによる解析を用いて検証し，第6章で結論を述べる．

2. 従来のスレッド管理システムとその問題点

今日までに，スレッド管理システムに関する研究は多く行われてきた．商用並列計算機が普及した1980年代に，Mach，V system，Firefly，Amoebaなどで用いられた初期のスレッド管理システム，スレッド切り替えを減らすための2レベルのスレッド管理およびその改善[2),6)~9),11),13),14),21)]，ユーザによるスレッド管理の自由度を向上させるfirst-class user-level threads[12),20)]，2レベルスレッド管理におけるユーザレベルのスレッド管理とカーネルのスレッド管理の間の不整合を解消するscheduler activations[1)]，プロセス間通信時に頻発するスレッド切り替えパターンを最適化するMachのcontinuation[5)]などが知られている．

これらの活発な研究にもかかわらず，従来のスレッド管理システムはいずれも，最適なスレッド切り替え動作以外の余分なスレッド切り替えを行っている．本章では，これら従来のスレッド管理システムにおいて，どのような余分なスレッド切り替えが起こっていたか，またその原因は何かを明らかにする．以下では，スレッド切り替えをさらに2種類の切り替えに分けて議論を進める．プロセッサの実行コンテクストを入れ換える**コンテクスト切り替え**と，特権モードと非特権モードとの間の切り替えである**ドメイン切り替え**である．

2.1 2レベルスレッド管理におけるスレッド切り替え

共有メモリ型マルチプロセッサでは現在，2レベルのスレッド管理と呼ばれる方法が広く用いられている．これはカーネルがプロセッサを多重化して実現した**カーネルスレッド**を，各プロセス内のユーザスケジューラがさらに多重化することによって**ユーザスレッド**をつくり，並列アプリケーションをユーザスレッド上で実行するものである．こうすることで，一つのプロセス内での，あるユーザスレッドから別のユーザスレッド

へのスレッド切り替えが，プロセス内部のコンテクスト切り替えのみで実現できる（すなわちカーネルへのドメイン切り替えを伴わない）という利点がある．

2レベルのスレッド管理では，各プロセスが自分の使用するカーネルスレッドを明示的にシステムコールで生成する．カーネルはそれらのカーネルスレッドをプレエンプティブにスケジュールするが，実際にどのカーネルスレッドにプロセッサが割り当てられているかをユーザスケジューラに知らせることはない．ユーザスケジューラとカーネルの間のこの機能の切り分けが原因で，次の二つの余分なスレッド切り替えパターンが発生する．

(1) 通常各プロセスは，カーネルスレッドを生成する際にシステム中の他のプロセスの存在を考慮しない．このため，実行可能なカーネルスレッドが，プロセッサの数より多くシステム内に生成される．この結果一つのプロセス内で，二つのカーネルスレッド間のコンテクスト切り替えが発生するが，これは余分なドメイン切り替えを含んでいる．このパターンの切り替えは本質的には，二つのユーザスレッド間のコンテクスト切り替えによって実現できるためである．

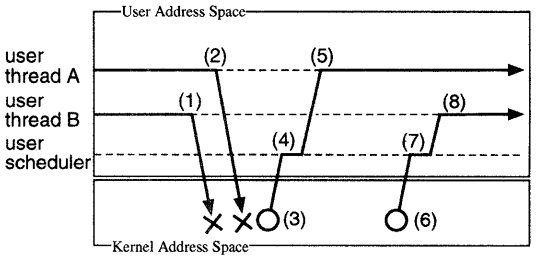
(2) ユーザスケジューラに，どのカーネルスレッドが実際に走行中かを知る方法が提供されないので，スピンロックが使えないという問題が発生する[18]．このためプロセス内での同期にはスリーピングロックが用いられることになり，同期ごとに必ず，本来不必要なコンテクスト切り替えが起こる．

(1)の問題は，各プロセスのカーネルスレッドの数を，プロセッサ資源の使用状況にあわせて調整する機構がないことが原因で発生している．また(2)の問題は，どのカーネルスレッドが走行中かを，カーネルからユーザスケジューラに効率的に知らせる機構がないことが原因で発生している．First-class user-level threads や，Mach の continuation を含む多くのスレッド管理システムは，これらの問題を含んでいる．

2.2 Scheduler activations におけるスレッド切り替え

Scheduler activations は，上記の二つの問題点を解決したシステムである．Scheduler activations は，各プロセスのカーネルスレッド数の合計を，システムのプロセッサ数と等しくするための機構を持っている☆．また，scheduler activations におけるユーザスケジューラは，どのカーネルスレッドが走行中であるかを知るための機構をもつ．一方，scheduler activations では，

☆ 同様のアイデアの概略は，文献19)にも示されている．



(1) スレッド B がカーネルにより停止される．(2) スレッド B が停止したことを通知するアップコールのため，スレッド A が停止される．(3) スレッド B がアップコールのために初期化される．(4) アップコールがユーザスケジューラに到着する．(5) ユーザスケジューラがスレッド A を再開する．(6), (7), (8) あらたなプロセッサがユーザスケジューラにアップコールの形で到着し，スレッド B が再開される．

図1 Scheduler activations における不必要なスレッド切り替えの例

Fig. 1 An example of unnecessary thread switching in the "scheduler activations" scheme.

これらの機構をアップコール[4]によって実現しているため，ドメイン切り替えが従来の2レベルのスレッド管理に比べても数多く起こるという欠点がある．

具体的な例をあげると，scheduler activations ではカーネルスレッドのプレエンプション時に，停止されるカーネルスレッドを持つプロセス（P と呼ぶ）に対してアップコールを用いた通知を行う（図 1）．ここで，アップコールに用いるために別のプロセッサが必要なため，scheduler activations では，プロセス P の別のカーネルスレッドを一旦停止させ，そのカーネルスレッドのプロセッサを用いて，二つのカーネルスレッドが停止したことをアップコールする．この結果，カーネルスレッドのプレエンプションを1回行うごとに，カーネルスレッドの停止と再開が2回ずつ起こることになる．さらに，この機構の実現にはプロセッサ間の割込みが必要となるため，カーネル内の構造，特に割込み処理ルーチン群の構造に大幅な変更が必要となる．

2.3 考　　察

従来のスレッド管理システムにおいて，上に述べた不必要なスレッド切り替えが起こっていた理由をまとめると，以下の3点に集約される．第1に，ユーザスケジューラがカーネルスレッドを明示的に生成していたため，カーネルスレッドの数の管理ができなかったこと．第2に，カーネルスレッドの状態に関する情報が，ユーザスレッドの管理に利用できなかったこと．第3に，ユーザスケジューラとカーネルとの間で，システムコールとアップコール（またはソフトウェア割り込み）という同期的なインタフェースを採用してい

たこと．これらを排除できれば，最適なスレッド切り替え動作を実現するスレッド管理システムができると考えられる．

3. 最適なスレッド切り替え動作を実現するスレッド管理システム

2章で行った考察をもとに，ユーザスケジューラとカーネルとが，非同期的なインタフェースを用いて協調的，相互支援的なスレッド管理を行う方法を提案する．本提案では，ユーザスケジューラとカーネルとが互いに情報交換をするとともに，相手の動作を適切に操作することを可能にする．また，この協調動作に際してユーザスケジューラとカーネルとが同期する必要がないように，共有メモリ領域の使用法を定める．

具体的には，提案するスレッド管理システムは以下の構成をとる．本システムは2レベルのスレッド管理システムと同様カーネルとユーザスケジューラによる2階層からなる．これらの間の協調動作を，以下の三つの要素によって行う．

Unstable カーネルスレッド: 本システムにおけるカーネルスレッドである unstable カーネルスレッドは，ユーザスケジューラの要求とは無関係に，カーネルによって生成，停止，再開，終了される☆．これによってカーネル自身がカーネルスレッド数の制御を行うことが可能となる．

共有メモリ領域 C-area: C-area はプロセスごとに一つ設けられ，カーネルとそのプロセスとの間で共有されるメモリ領域である．カーネルとユーザスケジューラとの間の協調動作は，ユーザが特に指定しない限りこの C-area を用いて行われる．ユーザスケジューラは，unstable カーネルスレッドと C-area の情報とを用いて，ユーザスレッドを実現する．

Notifier カーネルスレッド: 特別な方法でユーザスレッドのスケジューリングを行う場合などには，ユーザスケジューラとカーネルが同期的に協調動作することが不可欠な場面も考えられる．このため，ユーザスケジューラが指定したカーネル内イベントを同期的に通知する機構として，Notifier を設ける．Notifier カーネルスレッドは，ユーザが指定したイベントがカーネル内に発生した時点で生成される unstable カーネルスレッドである．

3.1 Unstable カーネルスレッド

カーネルスレッドの数の管理を適切に行うために，unstable カーネルスレッドは，ユーザスケジューラの直接の要求とは関係なくカーネルによって生成される．一旦停止した unstable カーネルスレッドは，カーネルまたはユーザスケジューラによって再開される[10]．Unstable カーネルスレッドが生成されると，そのスレッドはプロセスごとに定められたエントリアドレスから実行を開始する．さらに，カーネルによる unstable カーネルスレッドのスケジューリングは，プロセス単位の公平なプロセッサ割り当てのみを保証する．カーネルスレッド単位の公平性は保証されないので，unstable カーネルスレッドが一旦停止すると，任意に長い期間停止し続ける可能性がある．

この unstable カーネルスレッドの利点は，カーネルが各プロセスのカーネルスレッドの数を決定できる点と，カーネルスレッドの制御をユーザスケジューラとカーネルの間の同期的なやりとり，すなわちシステムコールやアップコールを一切なしで行える点である☆☆．前者は scheduler activations でも達成されていたが，後者は達成されていなかった．

カーネルがカーネルスレッドの数を決定できることは，システム全体でのカーネルスレッドの数を全プロセッサ数にもっとも近くするために重要である．これを実現するために，unstable カーネルスレッドのスケジューリングは以下のように行われる．カーネルは，あるプロセスに対してプロセッサの割り当てを行う時，またその時にのみ，そのプロセス内の unstable カーネルスレッドの生成または再開を行う．また unstable カーネルスレッドを終了または停止する場合には，必ずそのカーネルスレッドを実行していたプロセッサを別のプロセスへ割り当て直す．このように，カーネルスレッドのスケジューリングをプロセッサ割り当てと結合した結果，一つのプロセス内の二つのカーネルスレッドの間では，コンテクスト切り替えは起らなくなる．すなわち，カーネル内のコンテクスト切り替えは常にプロセス間のプロセッサ資源の再分配に伴うもののみとなる．

3.2 共有メモリ領域 C-area

C-area（図 2）はユーザスケジューラとカーネルとの間の双方向の通信チャネルとして機能する．カーネルが提供する情報は，unstable カーネルスレッドのスケジューリング状態（実行中，プロセス内で停止中，カーネル内で停止中のいずれか），および停止中の unstable カーネルスレッドの最終プロセッサコンテクス

☆ カーネルが各プロセスに割り当てるカーネルスレッドの数は時々刻々と変化するため，ユーザスケジューラからはカーネルスレッドの存在が不安定（unstable）に見える．

☆☆ 唯一の例外は，アプリケーションの並列性が急激に低下した際に，自発的にカーネルスレッドを終了させるためのシステムコールである．

```
enum status { RUNNING, BLOCKED, SUSPENDED,
              RECOVERING, RECOVERED };
/*
 * C-area (communication area between
 * the kernel and user scheduler)
 */
struct {
  struct {
    struct registers_t   c_registers;
    enum status          c_status;
  } c_kthread[MAX_KTHREAD];

  int c_max_parallelism;
} c;
```

図2 共有メモリ領域C-areaの定義
Fig. 2 The definition of the C-area.

トである．図2の中の配列c_kthreadの要素おのおのが，一つのunstableカーネルスレッドの状態を保持する．スケジューリング状態はc_statusに格納され，c_registersは最終プロセッサコンテクストを格納する．ユーザスケジューラが提供する情報は，現在実行可能なユーザスレッドの数c_max_parallelismである．カーネルはこの数字を越えない範囲でプロセッサをプロセスに割り当てる．

ユーザスケジューラとカーネルとのやりとりを非同期にするために，C-areaへのアクセスは以下の原則にのっとって行われる: 情報の送信者は情報が更新された時点でC-areaにそれを反映させ，情報の受信者は必要になった時点でC-areaを参照する．C-areaへのアクセス競合を避けるための排他制御は存在するが，ユーザレベルで動作しているスレッドとカーネルレベルで動作しているスレッドとが，C-areaを用いて同期することはない．

カーネルによるunstableカーネルスレッドの動作状態の通知: カーネルは，unstableカーネルスレッドのスケジューリング状態が変化する度に，対応するc_kthreadのc_statusとc_registersの更新を行う．Unstableカーネルスレッドが生成された時点でc_statusはRUNNING状態となる．これは，そのunstableカーネルスレッドがプロセッサを割り当てられていることを意味する．Unstableカーネルスレッドが I/O処理やページフォールトなどによって停止した場合，BLOCKED状態となる．また，カーネルによるプレエンプションによってunstableカーネルスレッドが停止した場合には，SUSPENDED状態になり，BLOCKED状態とは区別される．カーネルはc_statusをSUSPENDED状態にする直前に，unstableカーネルスレッドのプレエンプション時におけるプロセッサコンテクストをc_registersに代入する．c_statusがBLOCKEDまたはSUSPENDED状態にあるunstableカーネルスレッドは，プロセッサの再割り当てによってRUNNING状態に復帰する可能性がある（図3）☆．

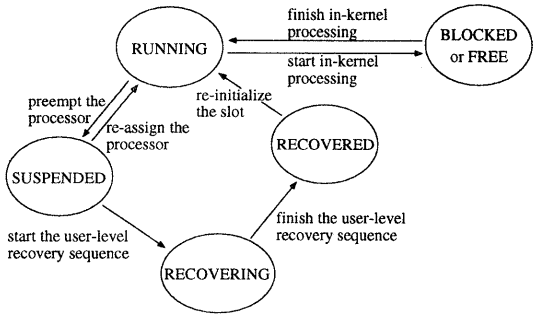


図3 Unstableカーネルスレッドの五つの状態
Fig. 3 States of an unstable kernel thread.

ユーザスケジューラはc_statusとc_registersを適宜参照し，適切な処理を行うことによってユーザスレッドが永遠に停止することを避ける．すなわちSUSPENDED状態にあるunstableカーネルスレッドを発見すると，そのunstableカーネルスレッドに対応するユーザスレッドの実行をユーザレベルで再開する．unstableカーネルスレッドがSUSPENDED状態にあるかどうかを点検するタイミングは，ユーザスレッドのスケジューリングポリシーによって異なるが，ここではノンプレエンプティブスケジューリングを例にとって話を進める☆☆．

ノンプレエンプティブスケジューリングでは，すべてのユーザスレッドが最終的に実行を終了すればよく，特定のタイミングで特定のユーザスレッドを実行する必要はない．このため，動作中のユーザスレッドAが別のユーザスレッドBと同期を行う時点にのみ，ユーザスレッドBを実行中のunstableカーネルスレッドがSUSPENDED状態にあるかどうかを点検する．

あるユーザスレッドを実行していたunstableカーネルスレッドが，SUSPENDED状態になった場合，それを再開するためには，c_registersをユーザスレッドのスレッドコントロールブロック（TCB）に代入して，そのTCBを実行待ちスレッドキューに加えるか，c_registersの内容を直接プロセッサのレジスタセットに代入すればよい．ユーザスケジューラはこの再開処理の後，このunstableカーネルスレッドの状態を

☆ 図3のFREE状態は，まだ割り当てられていないカーネルスレッド，および終了したカーネルスレッドがとる状態である．

☆☆ プレエンプティブスケジューリングや優先度つきスケジューリングの場合には，3.3節で述べるnotifierカーネルスレッドを契機としてこの点検を行う．

RECOVERED 状態に置くことによって，再開処理がユーザレベルで終了したことをカーネルに通知する．

C-area へのアクセスにおける競合の解消: C-area を通じた上記のユーザスレッドの再開操作を実現する際に問題となるのが，ユーザスケジューラとカーネルとの c_registers に対するアクセス競合が存在することである．これを防ぐために，カーネルとユーザスケジューラは，以下に述べるプロトコルにしたがって C-area へのアクセスを行う．

c_registers にアクセスする前に，ユーザスケジューラは不可分命令を用いて c_status を SUSPENDED から RECOVERING に変更し，これが成功した後で c_registers にアクセスする．c_status の書き換えが失敗した場合には，unstable カーネルスレッドの実行をカーネルが再開したことを意味するので，ユーザレベルの実行再開は不必要となる．一方，カーネルが c_status を SUSPENDED から RUNNING に変更する場合にも不可分命令を用いるとともに，RECOVERING 状態になった unstable カーネルスレッドは再開しないこととする．こうすることで，一つのカーネルスレッドを複数回再開するという誤動作を防ぐ．

ユーザスケジューラは，カーネルが C-area を再利用できるように，ユーザスレッドの再開処理が終了した時点で c_status を RECOVERING から RECOVERED に変更する．カーネルが RECOVERED 状態の unstable カーネルスレッドを再開する場合には，まずこのカーネルスレッドのコンテクストを初期化したうえで使用する．これは RECOVERED 状態の unstable カーネルスレッドに対応するユーザスレッドの実行再開が，ユーザレベルですでに行われているからである．

なお，ユーザスケジューラの誤動作または悪意によって，上記のプロトコルが破られる場合も考えられる．しかし，不当な動作によってユーザスケジューラがつくり出すことができるのは，RECOVERING 状態のカーネルスレッドを多数作成することのみである．このため，ユーザスケジューラが上記のプロトコルを守らない場合でも，システム全体の動作に悪影響はでない．

ユーザスケジューラによる実行可能ユーザスレッド数の通知: ユーザスケジューラは，実行可能なユーザスレッドの数を C-area に反映させる．これは，ユーザスレッドの生成，停止，再開，消滅のいずれかが行われた時に c_max_parallelism を更新することにより，ドメイン切り替えやコンテクスト切り替えなしで行われる．一方カーネルは，c_max_parallelism を適宜参照することによって，プロセスに対するプロセッサの割り当てを行う．プロセッサがカーネル内でアイドルになった時点で，カーネルは各プロセスの c_max_parallelism を参照し，c_max_parallelism を越えない範囲で新たなプロセッサをプロセスに供給する．この動作は，アイドルになったプロセッサ自身が割り当て先が決定するまで連続的に行う．こうすることで，実行可能なユーザスレッドがあるにもかかわらずプロセッサがカーネル内でアイドル状態になる，という状況を避けることができる．すなわち本システムでは，ユーザスケジューラとカーネルとが非同期的な協調動作をしているにもかかわらず，ハードウェアの持つ並列性を常に最大限発揮することができる．

ここで，悪意を持つユーザスケジューラは，c_max_parallelism を用いて必要以上のプロセッサを手に入れることが，一時的には可能である．しかし，このようなプロセスは中期的なプロセッサ割り当て戦略によって，優先度をだんだんと低くすることができるため，システム内のプロセッサを長時間独占することは不可能である．

3.3 Notifier カーネルスレッド

ユーザスケジューラがノンプレエンプティブなスケジューリング以外のポリシー，例えばプレエンプティブスケジューリングや優先度つきスケジューリングを行う場合，ユーザレベルの自発的なスケジューリングを行うための契機が必要となる．プレエンプティブスケジューリングでは，あるユーザスレッドが一定時間以上連続して走行したら，他のユーザスレッドに制御を渡す必要がある．優先度つきスケジューリングでは，I/O 処理やページフォールトなどの理由で停止していた高優先度のユーザスレッドが実行可能になると，低優先度のユーザスレッドを停止して高優先度のユーザスレッドを実行する必要がある．ユーザスケジューラがこれらの契機を知るには，カーネルの支援が必要である．

本提案では Notifier カーネルスレッドを導入してこれを解決する．Notifier カーネルスレッドは，ユーザが指定したイベントがカーネル内に発生した時点で生成される unstable カーネルスレッドである．Notifier カーネルスレッドは，発生したイベントに関する情報（具体的にはページフォールト処理の起動と終了，I/O 処理の終了，ユーザの設定したアラーム時刻の到来）を特定のレジスタ群に保持した上で，ユーザスケジューラを起動する．ユーザスケジューラはこれを契機として各々のスケジューリングポリシーで必要とされる処理を行うことができる．Notifier カーネルスレッドとそれ以外の unstable カーネルスレッドとの違いは，生成のタイミングと生成時のレジスタ群の値のみ

であり，生成された後はnotifierカーネルスレッドもunstableカーネルスレッドと全く同じ振舞いをする．これを利用して，ユーザスケジューラはイベント処理後のnotifierカーネルスレッドを，ユーザスレッドのスケジューリングに用いることができる．

4. スレッド切り替えの最適化

本章では，本論文のスレッド管理システムが，スレッド管理に伴う三つのスレッド切り替え，すなわちプロセス内のコンテクスト切り替え，カーネル内のコンテクスト切り替え，およびプロセスとカーネルの間のドメイン切り替えの回数を最小にしていることを示す．また一つ一つのスレッド切り替えを従来と同等のコストで実現する方法を示す．

4.1 スレッド切り替え回数に関する考察

以下の3種類のスレッド切り替えは，ユーザスレッドの切り替え要求（アプリケーションプログラマが記述したものでも，コンパイラが生成したものでもよい）と，カーネルによるプロセッサ割り当て戦略を実現するために不可欠である．

(1) マルチタスク環境におけるプロセッサ割り当て戦略では，プロセスごとのプロセッサの割り当てを変更するために，あるプロセスから別のプロセスへのスレッド切り替えが起こる．この場合，一つのプロセッサがプロセスの壁を1回またぐ度に，最低2回のドメイン切り替えと最低1回のカーネル内のコンテクスト切り替えが必要である．

(2) 多くの場合，各プロセスのユーザスレッド数はカーネルスレッド数よりも大きくなる．そのようなプロセスでは複数のユーザスレッドを一つのカーネルスレッドで実行することになり，あるユーザスレッドから別のユーザスレッドに制御が移るごとに，最低1回のユーザレベルのコンテクスト切り替えが必要となる．

(3) 外部割り込みと，プロセスからのシステムコールを処理する際にも，スレッド切り替えが起こる．1回のシステムコールまたは外部割込みに際して，最低2回のドメイン切り替えが起こる．

理想的にはこの3種類以外のスレッド切り替えは必要ない．第3の種類のスレッド切り替えについては，既存のスレッド管理システムでも理想状態が実現されているものが多い．半面，第1，第2の種類のスレッド切り替えでは従来，2章でふれた不必要なスレッド切り替えが行われていた．提案したスレッド管理システムではこれらが以下のように解決され，上記3種類のスレッド切り替えのみによってシステムが稼働する．

アドレス空間をまたがるスレッド切り替えの最適化: プロセスAからプロセスBにプロセッサが移動する場合を考える．まずプロセスA中のプロセッサが，ドメイン切り替えを1回行うことによりカーネルに入る．この時，このプロセッサに対応するunstableカーネルスレッドは停止状態になり，プロセスAのユーザスケジューラにはC-areaを通じてこの変化が通知される．カーネルに入ったプロセッサは，カーネル内のコンテクスト切り替えによりプロセスBへのアドレス空間の切り替えを行った後，プロセスBへのドメイン切り替えによってプロセスBでの実行を開始する．

最後のステップでは，二つの場合が考えられる．プロセスBに停止中のカーネルスレッドが存在する場合には，そのカーネルスレッドの実行を再開することによりプロセスBのユーザスレッドを再開することができる．一方，プロセスBのすべてのカーネルスレッドが走行中である場合，新たなunstableカーネルスレッドを生成することによりユーザスケジューラに制御を一旦移し，ユーザレベルで実行可能なユーザスレッドの再開を行う．

前者の場合には，プロセスB中でユーザレベルのコンテクスト切り替えはおこらない．また後者の場合，カーネルからユーザスレッドBへのドメイン切り替えの時点で，プロセッサのレジスタセットの初期化を行わずにユーザスケジューラに制御を移すことができる．このためこれに続くユーザレベルのコンテクスト切り替えは，プロセスAからプロセスBへのコンテクスト切り替えの一部であると見なすことができる．すなわち，プロセッサはドメイン切り替え2回とカーネル内のコンテクスト切り替え1回でプロセス間の壁をまたぐことができ，理想状態が実現されている．

アドレス空間内でのスレッド切り替えの最適化: 従来のスレッド管理システムにおける，アドレス空間内でのスレッド切り替えでは，ユーザスケジューラがカーネルスレッドの状態を知るためにシステムコールを用いる必要がある点が問題だった．一方提案する方式では，ユーザスレッドが停止中であるか否かを，C-area上の情報により数マシン語命令のみで確認できる．これを確認した後で，ユーザレベルのコンテクスト切り替えを1回行えばよい．すなわちこの場合も理想状態が実現される．

4.2 スレッド切り替えのコストに関する考察

前章で示した最適なスレッド切り替え動作に加え，ここでは個々のスレッド切り替え動作が従来と同等のコストで実現可能であることを示す．これは，提案する方式によって，スレッド切り替えに伴う総コストを削減できることを示すためである．

本提案において，従来のスレッド管理システムよりもコストがかかる可能性のある部分を考察すると，C-areaの更新動作があげられる．提案方式においては，毎回のスレッド切り替えに対応してC-areaを適切に更新する必要があるため，そのコストをゼロ，もしくは個々のスレッド切り替えの操作の一部を用いて実現する必要がある．本提案では，以下の二つの実現技術によりこれを達成する．

C-areaの更新コスト: C-areaの更新を従来のスレッド管理システムにおけるスレッド切り替え操作に融合することにより，これまでと同等のコストでC-areaの更新を含むスレッド切り替えを行う．C-areaの更新には，c_statusの更新，c_registersの更新，c_max_parallelismの更新があるが，c_max_parallelismとc_statusの更新は1もしくは2機械語命令で行うことができるので，これらのコストは無視できると考えられる．一方，c_registersの更新は20から30機械語命令を必要とするため，スレッド切り替えコストの中で無視できないものとなる．c_registersの更新は，カーネルがカーネルスレッドを停止させる際に行われるが，我々はこれを以下に述べる方法で行っている．

基本となるアイデアは，ドメインスイッチ時のカーネルスレッドのプロセッサコンテクストを，従来のようにカーネルスタックの先頭に格納するかわりにc_registersに格納することである．すなわち，c_registersを単にプロセスへの通知のためにのみ使われるデータ構造ではなく，カーネルスレッド管理用のデータ構造の一部とした．実装上は，C-areaをプロセスが書き込み可能なページとプロセスは読み込みのみ可能なページの二つに分割し，c_registersは後者のページに格納することにした．こうすることでカーネルは，c_registersを安全に使用することができる．カーネルスレッドがカーネルにドメイン切り替えをする場合には，「カーネルスタック上にユーザレベルのプロセッサコンテクストを保存する」という従来の動作のかわりに，c_registersにプロセッサコンテクストを保存する．こうすることで，C-areaの更新を含めた1回1回のスレッド切り替えのコストを従来のスレッド切り替えと同等のコストで実現している．

C-areaアクセスにおけるページフォールトの回避: C-areaが通常のページと同様ページアウト可能であるとすると，C-areaの参照によって潜在的にページングが引き起こされ，それに伴うスレッド切り替えが発生する可能性がある．また，カーネルがc_max_parallelismを参照する場合にページングが引き起こされると，プロセッサが長時間カーネル内でアイドル状態におかれるため，スレッド切り替えが増加すること以上の悪影響が出る．このため我々は，C-areaはページアウト不可能な領域とすることにした．なお，ページングを考慮していないシステムでは，プロセスとカーネルとの共有領域にこのような制限は必要なく，例えばPsyche[14]ではユーザスレッドのレディキュー全体をカーネルとの共有メモリ領域においている．しかしこのような選択はページングのある環境では困難である．

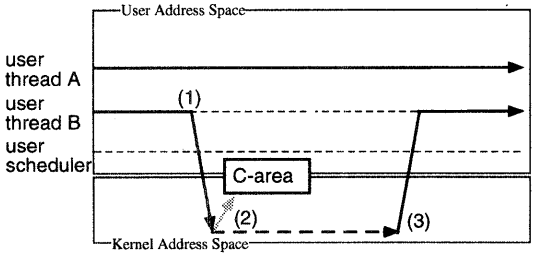
4.3 議　　論

Scheduler activationsとの比較: 本提案の新規性は，4.1節および4.2節で示したとおり，個々のスレッド切り替えのコストをほとんど変えることなく，最適なスレッド切り替え動作を実現した点である．これは，scheduler activations，Psycheなどを含む従来のスレッド管理システムでは達成されていなかった点である．

従来の方法の中でもっともスレッド切り替えコストを低減しているscheduler activationsと比較すると，本提案では，カーネルとユーザスケジューラとの間の協調動作を非同期的に行うか否かが根本的な新規点になっている．本提案ではできる限り（すなわち，ユーザが特に指定しない限り）非同期的な協調動作を用いるのに対し，scheduler activationsでは，同期的な協調動作のみですべてのスレッド管理に対応しようとしていた．この結果，2.2節であげた問題点が生じる他，以下の問題点も存在していた．

Scheduler activationsにおいてカーネルスレッドが停止すると，その時点で停止した事実がユーザスケジューラに通知され，その後の処理はスケジューラの責任となる．このため，カーネル自身がそのカーネルスレッドを再開することは不可能であった．この結果，カーネルスレッドを再開する処理が実際にはカーネルスレッドの初期化を伴うことになり，カーネルスレッドを生成する処理と同等のコストを伴う重い処理となっていた．これに対し本提案では，ユーザスケジューラの動作（この場合，ユーザスケジューラがカーネルスレッドを再開したかどうか）をカーネルが知ることができるので，停止したカーネルスレッドをカーネル自身が再開することが可能となった（**図4**）．

またscheduler activationsでは，ユーザスケジューラが通常必要としないイベント（ページフォールトの開始と終了など）までも，アップコールによって通知されてしまう．これに対し本提案では，非同期的に動作するunstableカーネルスレッドと，同期的に動作するnotifierカーネルスレッドを分離した．これによって，ユーザスケジューラが指定したイベントのみを同



(1) スレッドBがカーネルにより停止される.(2) スレッドBが停止したことがC-areaに書き込まれる.(3) カーネルがスレッドBを再開する.

図4 提案方式におけるカーネルスレッドの切り替えの例（図1との比較）

Fig. 4 An example of thread switching in the proposed scheme.

期的に授受し，そうでないイベントはより効率的な非同期的なインタフェースを用いて授受することが可能となった．また，notifierカーネルスレッドは，scheduler activationsのアップコールと本質的に同じ動作を行うものなので，実現コストも同じである．

共有メモリ領域の利用: カーネルとユーザスケジューラとの情報交換の手段として，共有メモリ領域を用いること自体は過去にも行われている．例えば，Ultra-UNIX[6]では，ユーザレベルでスピンロックを実現するために，カーネルとユーザ空間が共有メモリ領域を持つ．またPsycheでは，ユーザがカーネルスレッドの動作を細かく制御するなどの目的で，共有メモリ領域を用いていた．しかし，これらのシステムでは，カーネルスレッドの数がプロセッサ数よりも大きくなることによるスレッド切り替えの増加に対処できていなかった．この結果これらのシステムは，スレッド切り替え動作を最適化するという観点では，scheduler activationsより低い達成度となっている．

他のスレッド管理技術との連係: スレッド切り替え動作の最適化とは独立ではあるが重要な課題として，プロセッサ資源をどのようにプロセス群に割り当てるべきかという問題がある．特に共有メモリマルチプロセッサにおいて，一次キャッシュ，二次キャッシュをできるだけ再利用することの有効性は，文献17)などいくつかの研究によってすでに明らかにされている．本提案は，プロセッサ割り当て戦略については特に制限を設けていないので，これらの研究の成果と組み合わせて使用することができると考えられる．同じことが，並列プログラムの静的解析手法についてもいえる．

5. 実装と性能測定

前章までで提案したスレッド管理システムの有効性を調べるため，この機構の実装と性能測定を行った．実装は68030をプロセッサとするSONY NEWSワークステーション（クロック25MHz，主記憶8Mbyte）上のUNIXオペレーティングシステムのプロセス管理に変更を加える形で行った．実装の結果，ユーザスレッドおよびカーネルスレッドの基本操作のコストが明らかになった．**表1**は，この実装におけるユーザスレッドおよびカーネルスレッドの生成，停止，再開，コンテクスト切り替え，アドレス空間の切り替え，および消滅にかかる機械語命令数を計測した結果である．この計測はプロセッサのもつ命令トレースモードを用いて行った．

表1 スレッド基本操作のコスト（命令数）

Table 1 Cost of basic operations of threads (inst.).

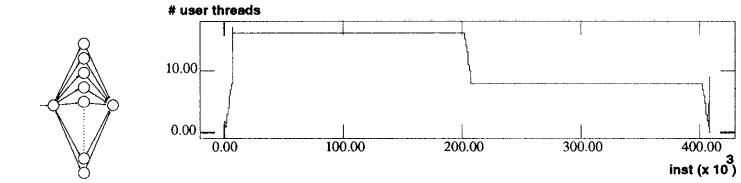
操作	ユーザスレッド	Unstable カーネルスレッド
生成	92	560
停止	13	60
再開	46	82
コンテクスト切替	40	260
アドレス空間切替	—	300
終了	46	513

この基本操作のコストを基に，既存のスレッド管理方式と本論文で提案している方式との比較を行った．単一の実装環境上の性能測定では，様々な環境や異なるワークロードにおけるこれらの方式間の差異を十分に検討することが困難であると考え，マルチプロセッサシステムのエミュレータを作成してこの上で各種のワークロードによる比較を行うことにした．

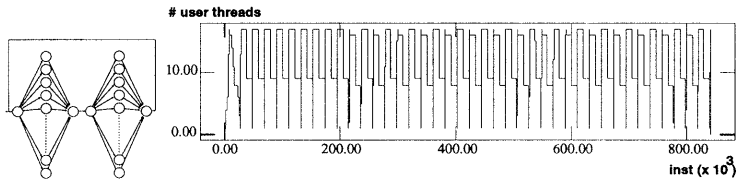
エミュレータは複数のプロセッサ，カーネルスレッド，ユーザスレッド，プロセスを実現しており，スレッド操作のコストには上の実装で得たユーザスレッドとカーネルスレッドの基本操作コストを用いている．このエミュレータにより，プロセスのユーザスレッドの計算，通信，同期とそれに伴うカーネルスレッドおよびプロセッサの挙動を，機械語命令数のレベルで再現することができる．一方，本エミュレータは一次および二次キャッシュの動作を考慮していない点，異なる機械語命令のコストをすべて同一としている点で制限がある．

5.1 比較するスレッド管理システム

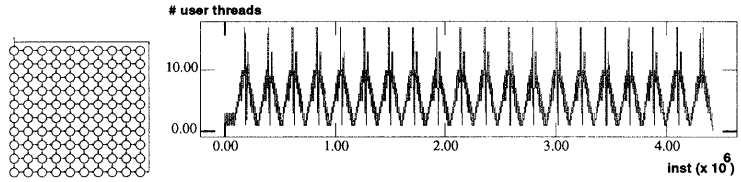
ここでは四つの異なるスレッド管理システムの性能を比較した．第1に，アプリケーションの実行にカーネルスレッドを直接用いるカーネルスレッドのみによる方法（図6，7中ではk-onlyと略記），第2にプロセスとカーネルとの間の情報交換および協調動作のない単純な2レベルスケジューリングの方法（図6，7中で



(a) MULT



(b) FORCE



(c) WAVE

図5 MULT, FORCE, および WAVE の性質
Fig. 5 The characteristics of MULT, FORCE, and WAVE.

表2 実験に使用した応用プログラムの性質
Table 2 Characteristics of application programs used in the experiments.

アプリケーション	典型的な同期パターン	平均並列度	平均同期間隔（命令数）
MULT	全ユーザスレッドによるバリア同期	7.86	1.7×10^5
FORCE	全ユーザスレッドによるバリア同期	7.85	1.7×10^3
WAVE	二つのユーザスレッドによる同期	4.10	9.0×10^2

は u/k two-level)，第3にアップコールによりプロセスとカーネルとが同期的な情報交換および協調動作を行う scheduler activations（図6，7中では scheduler activations)，そして第4に我々の提案する，共有メモリ領域を通じてプロセスとカーネルとが非同期的な情報交換および協調動作を行う方法（図6，7中では proposed scheme）である．

5.2 応用プログラム群

本実験では，3種類の通信，同期パターンを持つ並列アプリケーションを用いて，マルチタスク環境のワークロードを生成する．第1のアプリケーションは256×256の行列二つの積を求める MULT である．ここでは二つの行列をそれぞれ64×64のブロックに分割し，16のユーザスレッドを用いて計算を行う．プログラムの最後の部分でバリア同期を行ってプログラムを終了する．プログラム実行中はユーザスレッド同士の通信や同期はない．第2のアプリケーションである FORCE は，並列化した N 体問題である[3]．プログラムの各ループは二つのフェーズに分かれており，各フェーズで分割統治による並列化を行う．このプログラムも16のユーザスレッドを用いており，各フェーズの最初で全ユーザスレッドがバリア同期を行う．第3のアプリケーションはウエーブフロントの通信パターンを持つ WAVE で，10×10 のメッシュ状に配置されたユーザスレッドが，となりあう二つのユーザスレッドと通信を行う．

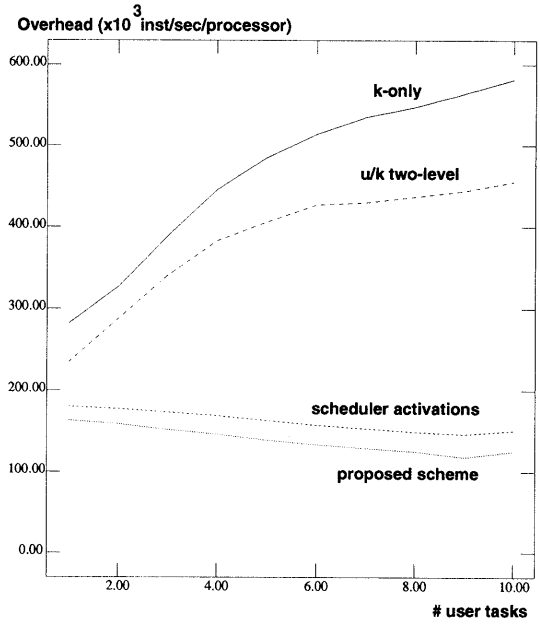
以下で個々のアプリケーション単独の性質を調べた．**図5** に三つのアプリケーションのユーザスレッド同士の通信，同期パターン（図5の左）と，プロセッサ数が8の場合におけるプログラム実行中の実行可能なユーザスレッド数のトレース（図5の右）を示す．MULT, FORCE, WAVE の典型的な同期のパターン，およびトレースを集計した得た平均並列度と平均バリア同期間隔（1プロセッサあたりの命令数）を**表2** にまとめた．

5.3 実験結果

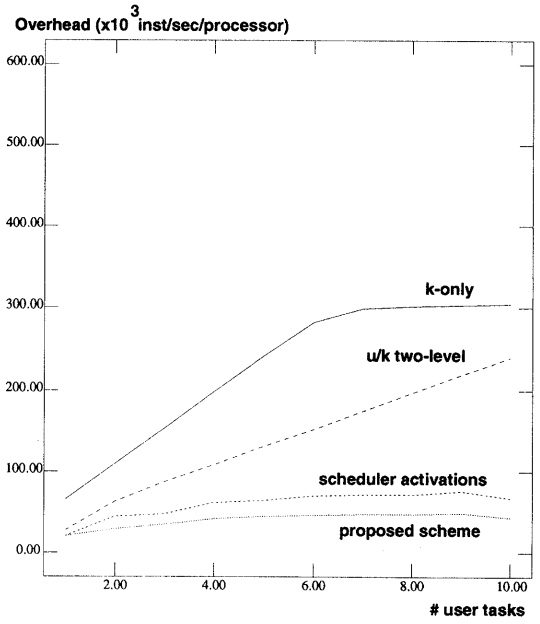
同時実行されるプロセス数（システムのロード）がスレッド管理システムの性能に与える影響を測定した．MULT，FORCE，WAVEの組合せにより，7種類の異なるワークロード（MULTのみ，FORCEのみ，WAVEのみ，MULTとFORCE，FORCEとWAVE，MULTとWAVE，MULTとFORCEとWAVE）で実験を行ったが，どれからも類似点のある結果を得た．そこでまずWAVEのみのワークロードでの結果で議論を進める（図6(a)）．グラフは，いくつかのプロセスが投入された場合の，プロセッサ1台1秒当たりのスレッド管理オーバヘッドを示している（X軸の単位はプロセス数，Y軸の単位は機械語命令数）．カーネルのみの方式と単純な2レベルスケジューリングでは，システムのロードが上がるに従ってスレッド操作コストも上昇している．これに対し，scheduler activationsと提案方式ではシステムのロードによらずほぼ一定のスレッド操作コストを示している．これは，前者2方式においてカーネルスレッド数が制御されていないために，カーネルスレッドのスケジューリングの際にアドレス空間の切り替えが起こる確率が大きくなることに起因する．Scheduler activationsと提案方式では，常に提案方式の方が低いスレッド操作コストを示している．これは，scheduler activationsにおけるアップコールによる不必要なドメイン切り替えが，提案方式では解消されていることに起因するものであると考えられる．提案方式によるスレッド管理コストの改善は，カーネルのみの方式に対して最大72.5%，単純な2レベルスケジューリングに対して最大59.5%，scheduler activationsに対して最大19.0%であった．

また，同期の回数がWAVEより少ないワークロード（例えばMULTのみ；図6(b)）を，WAVEのみのものと比べると，基本的なグラフの形に変化はないものの，いくつかの点で差異が見受けられる．第1に，scheduler activationsおよび提案方式におけるオーバヘッドが，それぞれWAVEの場合の半分程度になっている．これはアプリケーションから発生される同期の回数の差によるものである．第2に，カーネルスレッドのみによる方法では，ロードが8以上でのオーバヘッドの増加がない．これは，MULTではユーザスレッド間の同期がほとんど発生しないために，カーネルスレッドのプレエンプションによるオーバヘッドが，スレッド管理オーバヘッド全体のほとんどの部分を占めていることを示している．

第2の実験として，ここで用いた三つのアプリケーションとユーザスレッド間の通信パターンが類似のアプリケーションを想定して，各アプリケーションで計算と通信の割合を変化させた．計算と通信の割合を最初のものから1/16から16倍の間で変化させた．ここでも7種類の異なるワークロードで実験を行った結果，ほぼよく似た結果を得た．ここでは再びWAVE



(a) WAVEのみのワークロード



(b) MULTのみのワークロード

図6 スレッド操作コストとシステムのロードの関係
Fig. 6 The relationship between the system load and thread management cost.

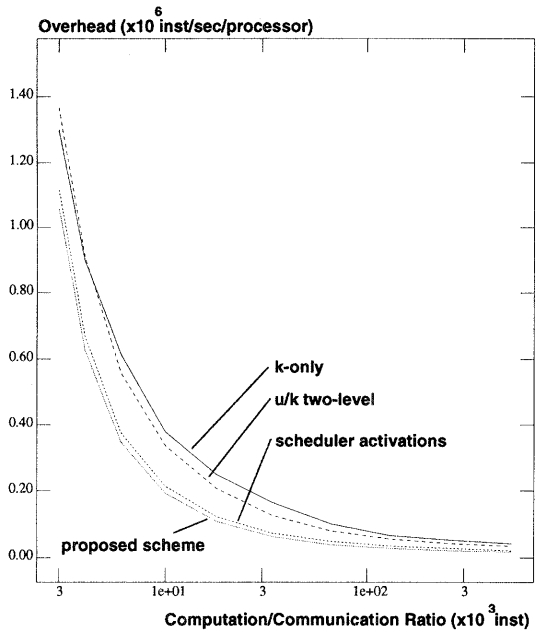


図7 スレッド操作コストと通信頻度の関係
Fig. 7 The computation ratio and cost of thread management.

による結果を示す（図7）．この図からわかるように，実験した範囲内では提案方式の他の方式に対する性能上の優位は，ユーザプログラムの通信頻度によって影響を受けない．

以上の実験の結果，提案方式では，カーネルのみのスレッド管理システム，単純な2レベルスケジューリング，scheduler activationsよりも低いオーバヘッドでスレッドを管理している．この結果は前節までで述べた提案方式の定性的な優位性を確認するものである．

6. まとめ

本論文では，プロセスとカーネルとが共有メモリ領域を通じた非同期な情報交換と協調動作を行うことにより，最適なスレッド切り替え動作を実現する方法を示した．スレッド管理システムは共有バス型並列計算機が商業的に成功した1980年代中ごろから研究と開発が行われてきた．特にこの論文の実験の中で比較に用いた三つの方式，カーネルスレッドのみの方式，単純な2レベルスケジューリング，そしてscheduler activationsは，より高性能のスレッド管理システムを探求する研究の進展に伴って順に現れてきた．実験による比較には用いなかった方式の中にもPsycheのfirst class user-level threadsやMachのcontinuationsなど，興味深い方式が数多く提案された．これら一連の研究は，高性能のスレッド管理システムを実現する方法を探求することを通じて，オペレーティングシステムの持つべき機能の選定，ユーザレベルのサービスとオペレーティングシステムカーネルのサービスの切り分け方といったオペレーティングシステムの構築方法を追求するものであった．本提案では，最適なスレッド切り替え動作を実現することによってこれらを性能上凌駕する方法を提案し，その効果を確認した．

今回の実験では，いくつかの面でまだ十分でない点が存在する．例えば，今回用いた実験では，二次キャッシュの効果やページングの効果が現実のシステムを反映していない．また，プロセッサ割り当て戦略と本提案との適切な相互作用によって，今回の結果以上にスレッド管理のコストを削減できる可能性もある．これらの点については，今後のより詳細な解析や，実際のアプリケーションを用いた実験を通して明らかにしていきたい．

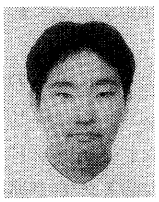
謝辞 実験環境であるSONY NEWSワークステーションとNEWS-OSを提供していただいたソニー株式会社およびソニーコンピュータサイエンス研究所の皆様に感謝いたします．

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 Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79 (1992).
2) Anderson, T.E., Lazowska, E.D. and Levy, H.M.: The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Comput.*, Vol.38, No. 12, pp. 1631–1643 (1989).
3) Barnes, J. and Hut, P.: A Hierarchical $O(N \log N)$ Force-Calculation Algorithm, *Nature*, Vol. 324, pp. 446–449 (1986).
4) Clark, D.: The Structure of Systems Using Upcalls, *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp.128–140 (1988).
5) Draves, R.P., Bershad, B.N., Rashid, R.F. and Dean, R.W.: Using Continuations to Implement Thread Management and Communication in Operating Systems, *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 122–136 (1991).
6) Edler, J., Lipkis, J. and Schonberg, E.: Process Management for Highly Parallel UNIX Systems, *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pp. 1–17

(1988).
7) Faust, J.E. and Levy, H.M.: The Performance of an Object-Oriented Threads Package, *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications and European Conference on Object-Oriented Programming*, pp. 278–288 (1990).
8) Golub, D., Dean, R., Forin, A. and Rashid, R.: UNIX as an Application Program, *Proceedings of the USENIX 1990 Summer Conference*, pp. 87–95 (1990).
9) Govindan, R. and Anderson, D.P.: Scheduling and IPC Mechanisms for Continuous Media, *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 68–80 (1991).
10) Inohara, S., Kato, K. and Masuda, T.: 'Unstable Threads' Kernel Interface for Minimizing the Overhead of Thread Switching, *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pp. 149–155, IEEE Computer Society (1993).
11) Inohara, S., Kato, K., Narita, A. and Masuda, T.: Thread Facility Based on User/Kernel Cooperation in the XERO Operating System, *ICICE Transactions on Information and Systems*, Vol. E75-D, No. 5, pp. 627–634 (1992).
12) Marsh, B.D., Scott, M.L., LeBlanc, T.J. and Markatos, E.P.: First-Class User-Level Threads, *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 110–121 (1991).
13) Powell, M.L., Klieman, S.R., Barton, S., Shar, D., Stein, D. and Weeks, M.: SunOS Multi-thread Architecture, *Proceedings of the USENIX 1991 Winter Conference*, pp. 65–79 (1991).
14) Scott, M.L., LeBlanc, T.J. and Marsh, B.D.: Multi-Model Parallel Programming in Psyche, *Proceedings of the Second ACM Symposium on Principles and Practices of Parallel Programming*, pp. 70–78 (1990).
15) Tevanian, A. Jr., Rashid, R., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W.: Mach Threads and the Unix Kernel: The Battle for Control, *Proceedings of the USENIX 1987 Summer Conference*, pp. 185–197 (1987).
16) Thacker, C.P., Stewart, L.C. and Satterhwaite, E.H. Jr.: Firefly: A Multiprocessor Workstation, *IEEE Trans. Comput.*, Vol. 37, No. 8, pp. 909–920 (1988).
17) Vaswani, R. and Zahorjan, J.: The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors, *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 26–40 (1991).
18) Zahorjan, J., Lazawska, E.D. and Eager, D.L.: The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, pp. 180–198 (1991).
19) 松本 尚: マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察, 情報処理学会計算機アーキテクチャ研究会報告, Vol. 79, No. 1, pp. 1–8 (1989).
20) 追川修一, 徳田英幸: 外部スケジューラに基づくマイクロカーネルの構成, コンピュータソフトウェア, Vol. 11, No. 5, pp. 387–399 (1994).
21) 新城 靖, 清木 康: 仮想プロセッサを提供するオペレーティング・システムの構築法, 情報処理学会論文誌, Vol. 34, No. 3, pp. 478–488 (1993).

(平成6年5月9日受付)
(平成7年7月7日採録)



猪原 茂和（正会員）

1967年生. 1989年東京大学理学部情報科学科卒業. 1991年同大学大学院修士課程修了. 1993年同博士課程中退. 1993年より東京大学大学院理学系研究科情報科学専攻助手. 分散・並列オペレーティングシステム, 永続データ管理システム, トランザクション処理などに興味を持つ. ACM, IEEE, USENIX各会員.

益田 隆司（正会員）

1939年生. 1963年東京大学工学部応用物理学科卒業. 1965年同大学大学院修士課程修了. 同年（株）日立製作所入社. 1977年から筑波大学, 1988年3月から東京大学に勤務. 現在, 同大学大学院理学系研究科情報科学専攻教授. 専門はオペレーティングシステム.