ユーザプログラムとカーネルの協調に基づくスレッドの設計と実現

岡坂史紀[†] 清水謙多郎[†]
芦原評[†] 亀田壽夫[††]

オペレーティングシステムのカーネルとユーザレベルプログラムの協調により実現される柔軟で効率のよいスレッド—マイクロプロセス—を設計し，実装を行った．マイクロプロセスの制御は基本的にプロセッサのユーザモードで実行されるため，スレッドの生成，切替え，同期，通信を，効率的かつ応用プログラムに適したスケジューリング方針に基づいて実行することができる．また，マイクロプロセス方式では，カーネルの記憶領域を多用しないので，一般にカーネルレベルのスレッドより1～2桁多いスレッドを生成することが可能である．さらに，マイクロプロセス方式では，従来のカーネルとユーザの協調によるスレッドの実現方式を推し進め，大域的な優先度に基づく横取り可能なスケジューリングを可能にしたほか，効率のよい遠隔手続き呼出し機構を提供している．本論文では，まず，従来のスレッドの実現方式の分類を行い，その問題点を示す．続いて，提案するマイクロプロセスの実現方式—カーネルとユーザレベルで統一的に適用されるスケジューリング機構，マイクロプロセスによる効率的な遠隔手続き呼出し機構，大域的なマイクロプロセスの識別方法—について説明する．最後に，マイクロプロセスの現在の実装に基づく性能評価の結果を示し，その有効性を明らかにする．

Design and Implementation of a Thread Mechanism Based on User/Kernel Cooperation

SHIKI OKASAKA,[†] KENTARO SHIMIZU,[†] HYO ASHIHARA [†] and HISAO KAMEDA [††]

This paper proposes a flexible and efficient mechanism for light-weight threads, called microprocesses, which is implemented through cooperation between the operating system kernel and the user-level run-time routines. Since most operations for microprocesses are executed in user mode, thread creation, deletion, context switch and inter-thread communication can be executed efficiently by using application specific scheduling and synchronization policies. Since microprocesses require only small amount of kernel space, thousands of threads can run concurrently. Several similar mechanisms have been proposed, but the mechanism of microprocesses is distinguished among the others in that 1. the kernel scheduler and the user-level scheduler are highly integrated; 2. microprocesses are scheduled based on common globally assigned priorities; and 3. it provides an efficient remote procedure call mechanism. In this paper, we first describe conventional implementation methods of threads. We then explain microprocess scheduling, remote procedure call mechanism and microprocess global identification. Finally, we discuss the current implementation and performance of microprocesses.

1. はじめに

スレッドは，従来のプロセスより粒度の細かい処理実体であり，ウィンドウシステムやオペレーティングシステムのサーバ，シミュレーション，知識処理などの応用プログラムにおいて，効率的な並行処理を実現するのに利用されている．

スレッドは，従来，オペレーティングシステムのカーネルもしくはコルーチンライブラリに代表されるユーザレベルのパッケージによって実現されてきた．

カーネルによる実現では，カーネルがスレッドを管理し，スレッドの生成，切替え，同期などの操作を実行する．このようなカーネルレベルのスレッドには次のような問題がある：

1. カーネルが主記憶上に管理情報を保持するため，多数の実体を生成できない．
2. スレッドの操作にカーネルが関与するため，ユーザ

† 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

レベルでの実現に比べて十分な効率が得られない．

3．仕様の変更・拡張に柔軟に対応できない．

ユーザレベルのスレッドでは，上記のカーネルスレッドにおける問題は解決されるが，横取り可能にしたり，入出力操作の実行やページ例外の発生によって並行性が失われないようにしたりすることが極めて困難であり，適用できる範囲も限定される．

最近では，カーネルレベルスレッドの問題を解決するいくつかの試みが行われている．Mach[1),11)]では，多数のカーネルスレッドの生成を可能にするため，カーネルの実行コンテクストを動的に割り当て，またその退避と回復をカーネルのプログラムが明示する機構（コンティニュエーション）を提案している．しかし，この機構は，上に述べたカーネルレベルスレッドのその他の問題に対して十分に対応していない．これに対して，オペレーティングシステムのカーネルとユーザプログラムが協調してスレッドを実現し，両者の利点を取り入れようとする試みがなされている[6),8)~10),12),13)]．このようなスレッドの実現方式をここでは**カーネル・ユーザ協調方式**あるいは単に**協調方式**と呼ぶ．協調方式では，大部分のスレッド制御をユーザレベルで行うため，高い効率と柔軟性を発揮することができる．しかし，従来の協調方式には，次のような問題がある：

1．カーネルとユーザパッケージの両方に存在するスレッドスケジューラが十分に統合されていないため，両スケジューラが共通して利用できるシステムレベルの大域的な優先度に基づいてスレッドをスケジュールすることができない．

2．アドレス空間を越える遠隔手続き呼出し機構に対して，カーネルとユーザの協調処理が与える影響について考慮していない．

本論文で提案する**マイクロプロセス**方式では，従来の協調方式をさらに推し進め，大域的な優先度に基づく横取り可能なスケジューリングを可能にしている．このため，プログラマがスケジューラの階層を意識する必要はなくなっている．さらに，アドレス空間を越える遠隔手続き呼出し機構に対し，カーネルとユーザによる協調処理が与える影響について検討を行い，カーネル・ユーザ協調方式に適した効率のよい遠隔手続き呼出し機構を提供している．

以下では，まず，従来のスレッドの実現方式の分類を行い，その問題点を示す．続いて，提案するマイクロプロセスの実現方式—カーネルとユーザレベルで統一的に適用されるスケジューリング機構，マイクロプロセスによる効率的な遠隔手続き呼出し機構，大域的なマイクロプロセスの識別方法—について説明する．最後に，マイクロプロセスの現在の実装と性能評価について述べる．

2．スレッド実現方式の分類

本章では，スレッドの実行コンテクストの管理方法から，従来のスレッドの実現方式の分類を試みる．スレッドの実行コンテクストは，ハードウェアレジスタの退避値，スレッドに関連したカーネルおよびユーザレベルのデータ構造体（スタック，実行状態や実行優先度などのスケジューリング情報を含む）からなる．一方，アドレス空間(ページ表)，ファイル表などスレッド間で共有されるものは，スレッドの実行コンテクストには含まれない．

以下では，スレッドの実行コンテクストを，カーネルが管理するカーネル実行コンテクストと，ユーザパッケージが管理するユーザ実行コンテクストとに分けて扱う．カーネルスタックはカーネル実行コンテクストに，ユーザスタックはユーザ実行コンテクストに含まれる．ハードウェアレジスタの退避値，スケジューリング情報は，スケジューラの実現される一方あるいは両方のレベルの実行コンテクストに置かれる．

図1は，カーネルレベルのスレッド（カーネルスレッド）の実現を示したものである．スケジューリングはカーネルレベルで行われ，スケジューリング情報はカーネル実行コンテクストに置かれる．ユーザ実行コンテクストは，カーネル実行コンテクストと一対一に対応する．伝統的なUNIX* のプロセスモデルは，一つのアドレス空間内にただ一つのカーネルスレッドが存在するというものである．

図2は，ユーザレベルのスレッド(ユーザスレッド)の実現を示したものである．ユーザスレッドの実行を制御するユーザレベルのスケジューラが存在し，そのためのスケジューリング情報はユーザ実行コンテクス

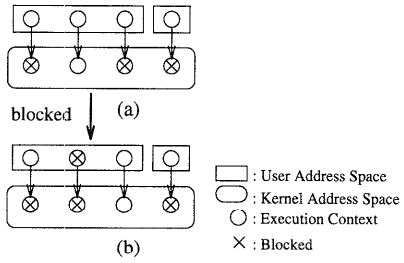


図1 カーネルスレッド
Fig. 1 Kernel threads.

* Unix は米国 X/Open Co., Ltd.がライセンスしている米国および他の国における登録商標です．

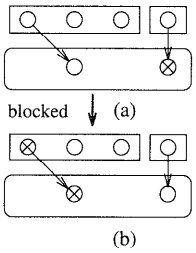


図2 ユーザスレッド
Fig. 2 User-level threads.

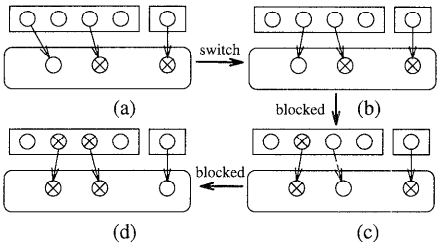


図3 カーネル/ユーザスレッドの組合せ
Fig. 3 User-level threads on kernel threads.

トに置かれる．ユーザスレッドの生成，破棄，切替えは，カーネルが関与しないので高速に行える．しかし，スレッドが封鎖型の入出力やページ例外によってカーネル内で封鎖されると，同じアドレス空間内の他のユーザスレッドは，たとえ実行可能であっても，実行を継続することができなくなる．ユーザスレッドの典型的な例として，従来 UNIX などで用いられてきたコルーチンライブラリがある[2]．また，こうした実現は多くの並行処理言語の実行系に見られる[3]．

図3は，カーネルスレッドとユーザスレッドを組み合わせて利用する方式を示したものである．カーネルスレッドは，ユーザスレッドを並行に実行する仮想プロセッサ（virtual processor）としてはたらく．この方式では，カーネルとユーザパッケージそれぞれに存在するスレッドのスケジューラは互いに独立している．応用プログラムでは，カーネル，ユーザ両方のスレッドを制御しなければならない．例えば，並行処理の効率を上げるため，入出力操作を頻繁に行う処理にカーネルスレッドを，計算を主に行う処理にユーザスレッドを使用するなどの工夫が不可欠である．また，カーネルスレッドがすべて（図3の例では2つ）封鎖されると，ユーザスレッドの実行を継続できなくなるという問題がある．

図4は，これをさらに発展させた，カーネルとユーザレベルの協調により並行処理を実現するカーネル・ユーザ協調方式を示したものである．この方式では，

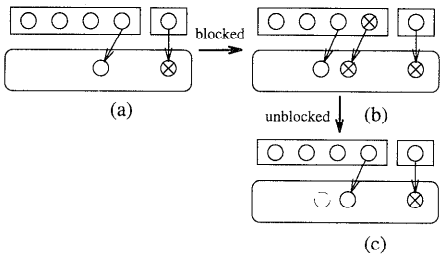


図4 カーネル・ユーザ協調方式
Fig. 4 Cooperation of kernel and user-level threads.

カーネルスレッドがユーザスレッドの実行に応じて動的に生成・破棄される．カーネル内のカーネルスケジューラとユーザパッケージ内のユーザスケジューラは，互いに協調してスレッドのスケジューリングを行う．例えば，カーネルスケジューラは，スレッドの実行を封鎖すると，ユーザスケジューラにそのことを通知する．これを受けてユーザスケジューラは，さらにカーネルスレッドを割り当てるようカーネルスケジューラに要求することができる．また，カーネルスケジューラがスレッドからプロセッサの制御を横取りする場合には，ユーザスケジューラを呼び出し，スレッドのハードウェアレジスタの値などを退避させたのち，カーネルスレッドを回収できる．この方式では，必要となるカーネルスレッドの総数は，カーネル内で封鎖されているスレッドの数に主に依存する．従って，ユーザが非常に多数のユーザスレッドを生成した場合にも，多くの場合，消費されるカーネルの記憶領域は少なく抑えられる．しかし，従来のカーネル・ユーザ協調方式では，ユーザスレッドそれぞれにシステムレベルの優先度を割り当ててスケジュールすることができない，また，カーネルスケジューラとユーザスケジューラの協調処理が，むしろシステムの実行効率を下げる場合があるという問題が残されている．本論文で提案するマイクロプロセス方式は，このような問題を改善するものである．

3. マイクロプロセスの実現方式

本章では，まず，カーネルによるマイクロプロセスの支援機構，ユーザスケジューラの役割，カーネル・ユーザ間の共有メモリインタフェースについて説明する．続いて，カーネルとユーザスケジューラの具体的な協調の例を示す．最後にマイクロプロセスのスケジューリング方針について議論する．以下では，マイクロプロセス方式によって実行されるユーザレベルのスレッドを，マイクロプロセスと呼ぶことにする．

3.1 カーネルによるマイクロプロセスの支援

3.1.1 カーネルスケジューラ

カーネルスケジューラは，カーネルスレッドの優先度に基づいたスケジューリングを行う．カーネルスレッドには，5つの状態が定義される．図5に，カーネルスレッドの状態遷移図を示す：

1. ユーザモード実行状態
2. カーネルモード実行状態
3. 実行可能状態
4. 休眠状態
5. 終了状態

実行可能状態のスレッドは実行待ち行列，休眠状態のスレッドは休眠待ち行列に登録される．

カーネルスレッドは，マイクロプロセスと結合している場合とそうでない場合がある．特に，カーネルは，どのマイクロプロセスとも結合していない実行可能なカーネルスレッドを各ユーザアドレス空間に対しプールすることができる．このため，カーネルは，プロセッサの制御をどのユーザアドレス空間に対し割り当てるかを，カーネルスレッドのスケジューリングによって決定することができる．

3.1.2 カーネルによるユーザスケジューラの支援

カーネルスケジューラは，スレッドの実行に応じて，ユーザスケジューラに対し事象通知を行う．カーネルが事象通知を行うと，事象の種別，事象を通知するカーネルスレッドと結合しているマイクロプロセスのハードウェアレジスタの値およびスケジューリング優先度を引数として，ユーザスケジューラが呼び出される．事象通知は，次のような場合に行われる：

- カーネルスレッドが休眠に入る場合

  ユーザスケジューラをアップコールし，事象Blockedを通知する．ユーザスケジューラからのアップコールの戻り値は，同一アドレス空間内にさらに実行可能なマイクロプロセスがあるか否か，さらに，もしあれば，そのスケジューリング優先度を示す．この場合，カーネルは，アドレス空間のカーネルスレッドプール内のスレッドを選択するか，新たにカーネルスレッドを生成し，指定された優先度を割り当て，再スケジューリングを行う．

- 現スレッドより優先度の高いスレッドが実行可能になった場合

  もし現スレッドより優先度の高いスレッドが実行可能であれば，カーネルスレッドがユーザモードに戻る際にユーザスケジューラから実行を再開するようスケジュールし，ユーザスケジューラに対して事象Preemptedを通知する．

- 休眠から目覚めたスレッドがユーザモードに戻る場合

  カーネルスレッドがユーザモードに戻る際にユーザスケジューラから実行を再開するようスケジュールし，ユーザスケジューラに事象Unblockedを通知する．

- カーネルで新たに割り当てられたカーネルスレッドおよびアドレス空間に対しプールされたカーネルスレッドがユーザモードに戻る場合

  カーネルスレッドがユーザモードに戻る際にユーザスケジューラから実行を再開するようスケジュールし，ユーザスケジューラに事象Resumedを通知する．この事象を通知するカーネルスレッドにマイクロプロセスは結合されていない．

- カーネルが実行優先度を変更したカーネルスレッドがユーザモードに戻る場合

  現スレッドの優先度を再計算し，カーネル内に優先度のより高い実行可能なスレッドがなければ，カーネルスレッドがユーザモードに戻る際にユーザスケジューラから実行を再開するようスケジュールし，ユーザスケジューラに事象Changedを通知する．

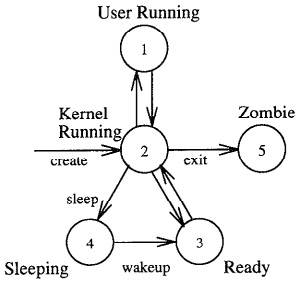


図5 カーネルスレッドの状態遷移図
Fig. 5 Kernel thread state transition diagram.

3.1.3 マイクロプロセスのためのシステムコール

カーネルは，次のようなシステムコールを提供する：

- initsched(void (**sched*)(void*))；

  ユーザスケジューラのアドレスをカーネルに通知する．このシステムコールを発行するまでは，応用プログラムの実行スレッドは1つに限られる．ユーザスケジューラは3.1.2項で述べたような事象の種別，マイクロプロセスの実行状態など（機種依存）を記述した構造体へのポインタを引数として呼び出される．

- set_concurrency(int *concurrency*)；

  アドレス空間に対しプールすることのできる，ど

のマイクロプロセスとも結合していない実行可能なカーネルスレッドの最大数を設定する．デフォルトでは1．並列プログラムでは2以上に設定する．

● yield(void)；

プロセッサの制御を放棄する．カーネルは，set_concurrency システムコールで指定したよりも多くの実行可能なカーネルスレッドがアドレス空間にあれば，システムコールを実行している現カーネルスレッドを破棄する．そうでなければ，ユーザアドレス空間のカーネルスレッドのプールに戻す．続いて，カーネルレベルの再スケジューリングを行う．また，カーネル・ユーザ間の共有メモリページ内の事象通知マスクをクリアする（3.3節参照）．

● void upcallreturn(int *prio*)；

カーネルからのアップコールを終了し，カーネルモードに戻る．*prio* には，アドレス空間内にさらに実行可能なマイクロプロセスがあればそのスケジューリング優先度を，そうでなければ−1を指定する．

3.2 カーネルからの事象通知に対するユーザスケジューラの処理

カーネルからの各事象通知を受けて，ユーザスケジューラが行うべき処理について以下に述べる：

●事象 Preempted の場合：

現マイクロプロセスより優先度の高いマイクロプロセスが実行可能になっている．ユーザスケジューラは，現マイクロプロセスの実行コンテクストを保存し，ユーザレベルの実行待ち行列に入れる．続いて，yield システムコールを発行し，プロセッサの制御を明け渡す．

●事象 Blocked の場合：

現マイクロプロセスが，カーネル内で封鎖されることを示す．ユーザスケジューラは，マイクロプロセスをユーザレベルの休眠待ち行列に入れ，現在のスタックポインタの値を保存する．続いて，upcallreturn システムコールを発行し，同一アドレス空間内に別に実行可能なマイクロプロセスがあるか否か，さらに，もしあれば，その優先度をカーネルに通知する．事象 Blocked では，カーネルスレッドとマイクロプロセスの結合は維持されたままである．

●事象 Unblocked の場合：

マイクロプロセスと結合しているスレッドが，カーネル内で休眠から目覚めたことを示す．ユーザスケジューラは，この事象を通知したカーネルスレッドと結合しているマイクロプロセスを休眠待ち行列から実行待ち行列に戻し，再スケジューリングを行う．休眠から目覚めたマイクロプロセスを休眠待ち行列内の中から識別するには，事象 Blocked を処理した際に保存しておいたスタックポインタの値を用いる．

●事象 Resumed の場合：

現アドレス空間にプロセッサの制御が割り当てられたことを示す．ユーザスケジューラは，事象を通知したカーネルスレッドに，実行待ち行列内のマイクロプロセスを結合する．

●事象 Changed の場合：

現スレッドの実行優先度がカーネルによって変更されたことを示す．ユーザスケジューラは，カーネルスレッドとマイクロプロセスとの結合を再スケジューリングする．

共有メモリ型マルチプロセッサ計算機においては，事象 Blocked および事象 Preempted が通知されたとき，現マイクロプロセスがスピンロックを獲得していれば，このロックをビジー待ちしている他のマイクロプロセスを中断させることにより，プロセッサ利用効率を高めることができる．

3.3 共有メモリインタフェース

各ユーザアドレス空間にはカーネルとの共有メモリ領域が取られる．ユーザスケジューラは，マイクロプロセスの優先度の変更をカーネルに反映させるため，また，カーネルによる事象通知を一時的に禁止するため，この共有メモリ領域を用いる．

3.3.1 実行優先度の通知

ユーザスケジューラが，制御を別のマイクロプロセスに切り替える場合には，新しいマイクロプロセスの優先度を共有メモリに書き込む．クロック割込みなどにより実行モードがカーネルモードに移ると，カーネルは，共有メモリに書き込まれている優先度をカーネルスレッドに反映する．この機構により，ユーザレベルのマイクロプロセスの切替えによって生じた優先度の変化を，システムコールを用いずにカーネルスレッドに反映させることができる．

ただし，ユーザスケジューラが，制御を優先度の低いマイクロプロセスに移す場合には，新しいマイクロプロセスの優先度を共有メモリに書き込み，直ちに yield システムコールを発行してカーネルレベルの再スケジューリングを行わせる．そうしないと，実行モードがカーネルモードに移るまでの間，より優先度の高いマイクロプロセスの実行を妨げる可能性がある．

3.3.2 事象通知の一時的な禁止

マイクロプロセス方式では，ユーザスケジューラが，一時的にカーネルからの事象通知を禁止することができる．この機構がなければ，カーネルがいつでも事象

を通知できるよう，ユーザスケジューラを再入可能にしなければならない．しかし，そのような実装は困難であり，応用プログラマが目的に応じたスレッドパッケージを作成できるようにするという意図にそぐわない．

事象通知の許可・禁止は，共有メモリページ内の事象通知マスクによって設定される．ユーザスケジューラは，事象通知を禁止したいとき，事象通知マスクをセットし，事象通知を許可するとき，事象通知マスクをクリアする．

ユーザスケジューラが通知を禁止している事象が発生した場合，カーネルは次のような対処を行う：

- 事象 Preempted, 事象 Changed の場合：
  カーネルは，事象通知マスクのある共有メモリページをユーザモードから書き込み禁止にし，そのままユーザスケジューラにユーザモードでの実行を継続させる．
- 事象 Unblocked, 事象 Resumed の場合：
  カーネルは，事象通知マスクのある共有メモリページをユーザモードから書き込み禁止にする．また，事象を通知するカーネルスレッドは，マスクが解かれるまで休眠あるいはビジー待ちによりカーネル内に留まる．
- 事象 Blocked の場合：
  カーネルは，事象 Blocked とそれに対応する事象 Unblocked を通知することなく，現カーネルスレッドを休眠させる．ユーザスケジューラは，事象 Blocked をマスクしている間に休眠に入ったか否かについて関知しない．

ユーザスケジューラが事象通知マスクをクリアするとき，すでに事象が発生していれば，事象通知マスクの置かれているページでページ例外が発生する．このとき，カーネルは，事象 Unblocked, 事象 Resumed を通知するため休眠しているカーネルスレッドを目覚めさせ，さらに，事象通知マスクをクリアしてから，共有メモリページをユーザモードから書き込み可能にする．この結果，プロセッサの実行モードがカーネルモードからユーザモードに戻る際に，事象通知を禁止していた間に発生した事象の通知がスケジュールされる．yield システムコールが発行された場合も，同様の処理が行われる．

事象通知マスクのクリアと事象通知のスケジュールを一括して行うため，書き込み禁止の事象通知マスクへのユーザモードからの書き込みを，カーネルが暗黙にマスクのクリアとして扱っていることに注意してほしい．従って，ユーザスケジューラが事象通知マスクをセットする場合は，既にマスクがセットされていないことを確かめてから，マスクを書き換える必要がある．

さらに注意すべきことは，ユーザスケジューラの実行がカーネル内で封鎖される場合があるということである．カーネルは，ユーザスケジューラが事象処理中に封鎖されたことを認識できるよう，事象通知の前に共有メモリのすべての事象通知マスクをセットする．従って，ユーザスケジューラの実行がカーネル内で封鎖された場合には，そのユーザスケジューラを含むアドレス空間はユーザスケジューラの封鎖が解かれるまで並行処理は再開されない．

3.4 信頼性の低いユーザスケジューラへの対策

信頼性の低いユーザスケジューラは，次のような場合，システムの動作に問題を引き起こす：

1. 事象通知を禁止したまま，元に戻さない．
2. 事象 Blocked のためのカーネルからのアップコールに応答しない．
3. 事象 Preempted のあと，yield システムコールを発行せず制御を明け渡さない．

これらの問題は，事象が発生した際，カーネルがウォッチドッグタイマを設定し，不当に長い事象通知禁止を監視することで検出可能である．また，そのような場合，強制的に問題を起したプログラムを終了させたり，システムレベルの再スケジューリングを実行するような対処が可能である．

また，プロセッサの制御を独占しようと意図的に高い優先度で実行し続けようとするマイクロプロセスに対しては，カーネルレベルで，各スレッドに割り当てられた優先度に基づいてスケジューリングを行うだけでなく，アドレス空間を単位として割り当てるプロセッサ時間を制限することで対処することが可能である．

3.5 カーネルとユーザスケジューラの協調の例

図 6 は，カーネルとユーザレベルとの間の制御の流れの一例を示したものである．マイクロプロセスが封鎖型の入出力操作を実行し，カーネル内で入出力処理の完了待ちに入るとする(1)．このとき，カーネルはユーザスケジューラに事象 Blocked をアップコールにより通知する(2)．ユーザスケジューラは，現在のマイクロプロセスを休眠待ち行列に入れ，制御をカーネルに戻す(3)．カーネルは，アップコールの戻り値から，同一アドレス空間内にさらに実行可能なマイクロプロセスがあるかどうかを調べ，もしあれば，新たなカーネルスレッドを生成する．入出力の完了待ちに入ったカーネルスレッドは休眠に入り，再スケジューリング

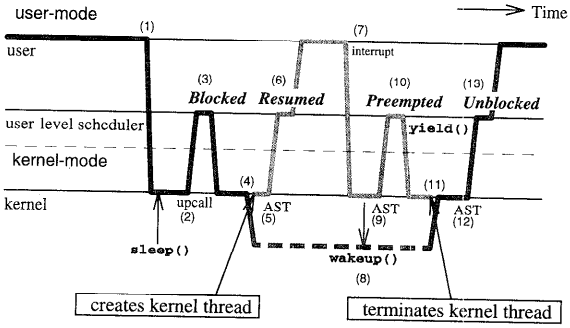


図6 カーネル/ユーザレベル間の制御の流れ
Fig. 6 Interaction between kernel and user-level schedulers.

が行われる(4).新たに生成されたカーネルスレッドがプロセッサの制御を獲得し,ユーザモードに移るとき,ユーザスケジューラに対し事象 Resumed を通知する(5).図中では,新たに生成されたスレッドの実行を薄い線で,休眠中のスレッドを点線で区別して示している.事象 Resumed を通知されたユーザスケジューラは,次に実行すべきマイクロプロセスを実行待ち行列から選び,そのユーザ実行コンテクストをロードする(現在のカーネルスレッドに結合する)(6).この結果,マイクロプロセスの実行が継続される.後に,装置入出力の終了により割込みが発生し(7),制御がカーネルに移ると,休眠していたカーネルスレッドを起こす(8).起こされたカーネルスレッドの優先度が,現在のマイクロプロセス(すなわち,これを実行しているカーネルスレッド)より高ければ,事象 Preempted をユーザスケジューラに通知する(9).ユーザスケジューラは,現在のマイクロプロセスを実行待ち行列に戻し,そのユーザ実行コンテクストを保存してから,yield システムコールを発行し制御を明け渡す(10).カーネルは余分になったカーネルスレッドを回収し,再スケジューリングを行う(11).休眠から起こされたカーネルスレッドは,ユーザモードに戻るとき,ユーザスケジューラに事象Unblockedを通知する(12).ユーザスケジューラは,休眠待ち行列内のマイクロプロセスを実行待ち行列に戻した後,Resumed の場合と同様に,次に実行すべきマイクロプロセスを実行待ち行列から選び,そのユーザ実行コンテクストをロードする(13).マイクロプロセスを実行中にページ例外が発生して待ちに入り,例外が解消されたときも,上と同様の処理が行われる.

3.6 マイクロプロセスのスケジューリングに関する議論

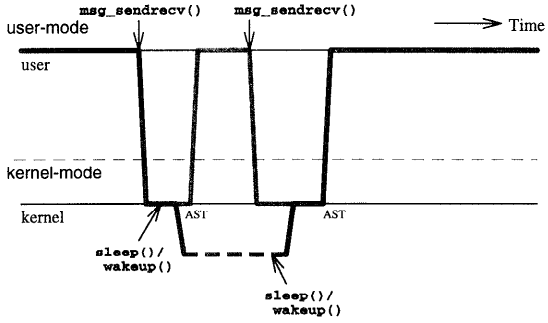
マイクロプロセスのスケジューリング機構は,ユーザレベルからマイクロプロセスの大域的な優先度を指定できるため,さまざまなスケジューリング方針に適用することができる.また,カーネルは,カーネルレベルのスケジューリング方針を事象 Changed を通知することによって,ユーザレベルに反映させることができる.現在の実装では,カーネルレベルで,固定優先度順のほかに多重フィードバック待ち行列によるスケジューリング方針を提供している.以下では,現在のスケジューリング機構を採用するまでの,筆者らの経験について述べる.

最初のマイクロプロセスの実装では,マイクロプロセスがカーネル内で封鎖され,新たにカーネルスレッドを生成する必要が生じたとき,その優先度を封鎖されたカーネルスレッドと同じに設定するという方針を用いていた.またカーネルスレッドの優先度の調節は基本的にカーネルが行っていた.これは,装置入出力を頻繁に行う I/O 束縛のスレッドに対しては優先度を高めに,計算を主体とした CPU 束縛のスレッドには優先度低めに調節していくというものであった.問題は I/O 束縛のマイクロプロセスと CPU 束縛のマイクロプロセスが混在する応用プログラムにおいて顕著に表れた.この方針では,次のようなことが繰り返された.

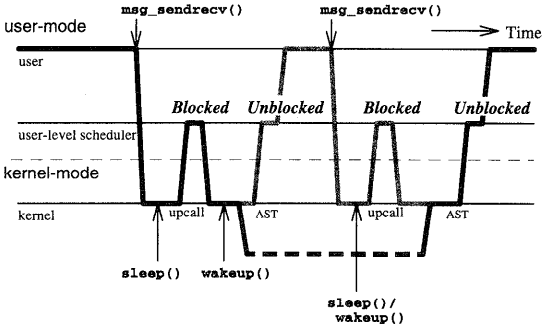
1. I/O 束縛のマイクロプロセスが封鎖され,CPU 束縛のマイクロプロセスが I/O 束縛のマイクロプロセスの高い優先度を継承して実行を再開する.
2. CPU 束縛のマイクロプロセスの優先度がカーネルによって下げられるよりも早く,I/O 束縛のマイクロプロセスの封鎖が解ける.

結果として,CPU 束縛のマイクロプロセスが I/O 束縛のマイクロプロセス並みの高い優先度で動作し続けることになり,システムのスループットの極端な低下を招いた.

そこで次に用いたのが,新たに生成するカーネルスレッドの優先度をあらかじめ低く設定するという方式であった.I/O 束縛のマイクロプロセスは,結合しているカーネルスレッドが横取りされるよりも先にカーネル内で休眠に入る確率が高いため,特定のカーネルスレッドと長期にわたって結合する傾向にある.従って,カーネルによる優先度の調節を反映して,高い優先度でスケジュールされるようになる.一方,CPU 束縛のマイクロプロセスは,頻繁にカーネルスレッドが横取りされ,制御を獲得するたびに優先度の低い新しいカーネルスレッドと結合するため,低い優先度でスケジュールされる.この方式では,ユーザスケジューラが I/O 束縛のスレッドから CPU 束縛のスレッドへの切替えを避ければ,本来の意図に合った優先度の調整が



(a) カーネルスレッド間の遠隔手続き呼出し
(a) Remote procedure call between kernel threads.



(b) 協調方式での遠隔手続き呼出し
(b) Kernel/User cooperation in remote procedure call.

図7 スレッド間通信での制御の流れ
Fig. 7 Control flow for inter-thread communication.

ある程度可能であった．しかし，実時間処理などより厳密な優先度に基づくスケジューリングを必要とする応用には，このようなアドホックな方針では対処できないのは明らかであった．

4. マイクロプロセスの遠隔手続き呼出し機構

本章では，アドレス空間を越える遠隔手続き呼出し機構に対して，カーネルとユーザによる協調処理が与える影響について検討を行い，マイクロプロセスを応用した効率のよい遠隔手続き呼出し機構の実現について説明する．

カーネル・ユーザ協調方式のスレッドでは，アドレス空間を越える遠隔手続き呼出しを行った場合，ユーザスケジューラにおける事象の処理が必要となり，従来のカーネルスレッドの場合と比べオーバヘッドが増大するという問題が生じる．**図7** (a) および図7 (b) は，それぞれ，従来のカーネルスレッドとカーネル・ユーザ協調方式のスレッドによる遠隔手続き呼出しの実行の様子を示している．図中，基本操作命令msg_sendrecv は，呼出し側(クライアント)では要求を送信し，応答を待つという処理を，呼び出される側（サーバ）では要求に対する応答を返し，次の要求を待つという処理をそれぞれ一括して行う命令とする．

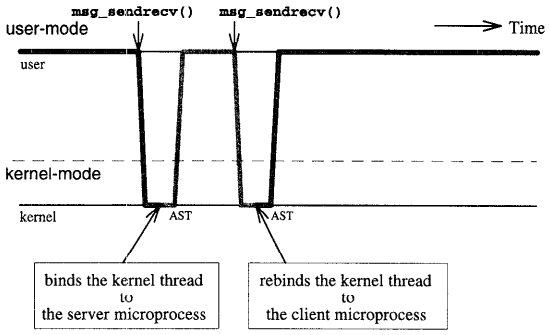


図8 マイクロプロセスによる遠隔手続き呼出し
Fig. 8 Remote procedure call between microprocesses.

図の例では，カーネル・ユーザ協調方式 (b) はカーネルスレッド (a) に比べ，1回の遠隔手続き呼出しで実行モードの切替えが2回多く発生している．カーネル・ユーザ協調方式では，事象 Blocked のアップコールに対しカーネルスレッドを追加するようユーザスケジューラがカーネルに要求すれば，カーネルとユーザ間のインタラクションはさらに増える．

この問題に対処するため，筆者らは，**図8**に示すように，クライアントのマイクロプロセスと結合していたカーネルスレッドをそのままサーバのアドレス空間に移動し，サーバのマイクロプロセスに結合させるという方式を提案する．この方式では，遠隔手続き呼出しを必要最小限のプロセッサの実行モード切替えによって実行できる．また，サーバの手続きはクライアントのスケジューリング優先度で実行される．

サーバとクライアント間の通信は，カーネル内に割り当てられたゲートを介して行われる．ゲートは，サーバとクライアント間をカーネルスレッドが移動するための中継場所であり，そこにマイクロプロセスのユーザモードにおけるハードウェアレジスタの退避値(以下では，単にユーザレジスタと略す)を保存することができる．以下では，次の3つの処理について説明する：

1. サーバが最初にクライアントからの要求待ちに入るときの処理．
2. クライアントが要求を発行し，サーバが受信するまでの処理．
3. サーバが応答を返し，クライアントに結果が戻るまでの処理．

サーバが最初に msg_sendrecv を発行すると，カーネルは，ゲートを作成し，呼び出したサーバマイクロ

プロセスのユーザレジスタをゲート内に保存する．続いて，カーネルは，サーバのユーザスケジューラに対し事象 Preempted を通知する．

クライアントが要求を発行したとき，ゲート内にサーバマイクロプロセスのユーザレジスタが保存されていれば，カーネルは，このユーザレジスタと，カーネルスレッドの実行コンテクストに退避されているクライアントマイクロプロセスのユーザレジスタとを取り替える．続いて，カーネルスレッドは，サーバのアドレス空間に移動し，ユーザモードに戻る．この結果，サーバにおいて受信操作の実行が継続される．

サーバは，要求を処理し終えると，結果をクライアントに戻すため，再び msg_sendrecv を発行する．今度は，カーネルは，サーバマイクロプロセスのユーザレジスタと，ゲート内のクライアントマイクロプロセスのユーザレジスタとを取り替える．続いて，カーネルスレッドは，クライアント側のアドレス空間に移動し，ユーザモードに戻る．この結果，クライアントは要求に対する結果を受け取る．

5. マイクロプロセスの識別

マイクロプロセスのように多数のスレッドを容易に生成できるようになると，例えば，ファイルサーバにおいて各ファイルごとに別々のスレッドを割り当てるというようなプログラミングが可能である．従って，マイクロプロセスをシステム内で大域的に識別する機構があると便利である．これに対し，従来の UNIX プロセスモデルを基礎とするスレッドは，プロセス（あるいはタスク）内部に定義され，これを外部のプロセスから識別することは不可能であった．

本方式では，マイクロプロセスの識別子をアドレス空間の識別子とアドレス空間内の識別子の対で表すこととし，アドレス空間の識別はカーネル，アドレス空間内のマイクロプロセスの識別はユーザレベルで行う方式を用いている．マイクロプロセスの実行の中断・再開などの操作において，対象となるマイクロプロセスの大域的な識別子が指定されると，まずカーネルによってアドレス空間が特定され，次にそのアドレス空間のマイクロプロセスパッケージによってマイクロプロセスが特定される．アクセス制御はアドレス空間に対してのみ適用される．このような方式を採用することにより，カーネルがすべてのマイクロプロセスの識別子を保持し識別を行うことによる，カーネルのメモリ領域の圧迫，効率の低下を回避できる．

6. 実　　装

6.1 動作環境

本方式を，筆者らが開発したオペレーティングシステム 006 に実装した．オペレーティングシステム 006 のカーネルは，マルチスレッド，仮想記憶，メモリマップドファイル I/O などの機能を備え，i 486 PC 上で稼動している．

006 カーネルでは，多くの UNIX や Mach と同様に，カーネルモードでの横取りがない．すなわち，カーネルモードで動作中のスレッドはプロセッサの制御を横取りされることはなく，スレッドの切替えは，スレッドの実行モードがカーネルモードからユーザモードに戻る直前に発生する非同期システムトラップ (AST) のときまで遅らされる．また，カーネル内での同期には，UNIX と同様な，現スレッドを休眠させる sleep と指定したスレッドを起こす wakeup の 2 つの同期基本操作を用いている[4]．

6.2 事象通知機構の実現

006 カーネルのマイクロプロセス支援のための拡張は，sleep ルーチンと AST ハンドラに集中した．新たに追加したコードは，C 言語で数百行程度であった．

事象 Blocked を除く事象の通知には，AST ハンドラで，カーネルスレッドに通知すべき事象があるかどうかを調べ，もしあればユーザモードでユーザスケジューラから実行を再開するようスケジュールするという方法をとった．カーネルが事象を通知するには，まず，その事象に対応したフラグをカーネルスレッドの実行コンテクスト内に設定する．そしてそのスレッドがカーネルモードからユーザモードに戻る直前に，AST ハンドラが事象に対応するフラグを検出すると，事象種別，ハードウェアレジスタの退避値を引数に，ユーザスケジューラが呼び出されるようスケジュールされる．**表 1** は，AST で調べられるフラグ（AST フ

表 1　AST フラグと事象の関連
Table 1　AST flags and relevant events.

AST フラグ	事象	フラグ設定
BLOCKED	Preempted	より優先度の高いカーネルスレッドが実行可能になったとき．
SLEPT	Unblocked	休眠に入るとき．
RECALC	Changed	スケジューリング優先度が変更されたとき．
PREEMPTED	Resumed	新たに生成されたときと，スレッドのプールに入れられたとき．

ラグ）と対応する事象，およびフラグが設定されるタイミングをまとめたものである．

事象 Blocked は，カーネルの sleep ルーチンの中からアップコールによりユーザスケジューラに通知される．カーネルスレッドが sleep を呼び出して休眠する場合，sleep ルーチンは，まず現マイクロプロセスに結合しているカーネルスレッドにフラグ SLEPT を立て，ユーザスケジューラに対しアップコールにより事象 Blocked を通知する．制御がカーネルに戻ると，sleep ルーチンの続きが実行される．もし，ユーザスケジューラの処理中に休眠していれば，sleep ルーチンは直ちに終了する．

事象 Unblocked と事象 Preempted は，1つのカーネルスレッドで同時に発生することがある．これは，カーネル内で封鎖されていたマイクロプロセスが起こされてプロセッサを獲得したが，ユーザレベルに戻る前に，より優先度の高いスレッドが続けて起こされた場合などに生じる．このとき，カーネルは，前者のマイクロプロセスに対し事象 Unblocked と事象 Preempted を2つ通知する代わりに，事象 Preempted だけを通知する．これをユーザレベルでは「封鎖解除後ただちに横取り」という事象として扱う．すなわちマイクロプロセスを休眠待ち行列から実行待ち行列に移し，ただちに yield システムコールを発行する．

6.3 マイクロプロセスパッケージの実現

今回，性能評価に用いた簡単なマイクロプロセスパッケージは，全体でも C 言語とアセンブリ言語で約600行という大きさである．このパッケージは，006カーネルのカーネルスレッドのスケジューリング基本操作ルーチンをコルーチンライブラリとして書き改め，さらに事象ハンドラを追加する形で実現した．

7. 性能評価

以降では，従来のカーネルスレッドと比較しながら，i 486 SX（16 MHz）の PC 上におけるマイクロプロセスの性能について述べる．

表2は，空のフォーク（空の手続きを実行して終了するスレッドを生成し，その終了を待つ）時間について，マイクロプロセス，カーネルスレッド，および新しいアドレス空間とカーネルスレッド，それぞれの場合を示している．カーネルスレッドの場合，スレッドの生成，再開，終了，終了待ちのために4回のシステムコール呼出しが行われる．新しいアドレス空間とカーネルスレッドの場合には，まず新しいアドレス空間を生成し，実行可能ファイルをマップする．その後で，カーネルスレッドの生成，再開，終了，終了待ちを行う．この処理には6回のシステムコール呼出しが行われる．実行可能ファイルの内容はあらかじめページキャッシュに保持された状態にしてあるので，この間ディスク I/O は行われていない．この結果はカーネル内の資源割当て，解放に要する時間的なコストを示している．

表3は，要求，受信，応答からなるメッセージ通信に要する時間を同一アドレス空間内のマイクロプロセスの場合とカーネルスレッドの場合とで比較したものである．この例では，各基本操作命令のコードに関してマイクロプロセスとカーネルスレッドで同等のものを使用している．従って，この結果は，システムコールを使ってスレッドの同期処理を行った場合のコストを示している．

表4は，プロセスフィルタを使って700までの素数を計算した例を示している．この例では126個のスレッドを必要とし，9,000回以上のメッセージの送受信，そして少なくとも18,000回のスレッドの切替えが行われる．表の Kernel thread（mutex and cv）では，スレッドの同期に006カーネルが標準で提供する相互排除ロック（mutex）と条件変数（cv）を使った場合の時間を示している．Kernel thread（message passing）では表3に示した同期型メッセージ通信基本操作命令を使った場合の時間を示している．マイクロプロセスは，応答時間でカーネルスレッドを使った実現と比べて2倍以上の性能を発揮している．さらに，126個

表 2 空のフォーク時間
Table 2 Null fork time.

	時間（μsec）
Microprocess	78
Kernel thread	328
Address space, Kernel thread	2,990

表 3 同期型メッセージ通信
Table 3 Synchronous message passing.

	時間（μsec）
Microprocess	95.1
Kernel thread	218

表 4 素数の計算
Table 4 Computation of prime numbers.

	時間（sec）
Microprocess	1.16
Kernel thread（mutex and cv）	8.79
Kernel thread（message passing）	2.32

のマイクロプロセスすべてがただ1つのカーネルスレッドを共有するため，カーネル空間の利用が1/126で済んでいる．

遠隔手続き呼出しにおいてカーネル内でのスレッド切替えを不要にすることによる性能の向上については，i 386 SX (16 MHz) PC上において，空の遠隔手続き呼出し時間を測定したところ，通常の遠隔手続き呼出しが393 μsecであったのに対して，4章で提案した方式に相当する遠隔手続き呼出しは61.9 μsecという結果を得ている．

8. 関連研究

カーネルとユーザプログラムで協調してスレッドを実現しようとする試みは，これまでいくつか提案されているが[6),8)~10),12)13)]，ページ例外の場合などあらゆる場合にスレッドの並行実行を可能にした機構としては，Scheduler Activations[10)] と Sun OS 5.0[12)] が挙げられる．

Scheduler Activationsでは，並列計算機におけるプロセッサの空間分割に基づいてスケジューリング機構が設計されているため，マイクロプロセス方式のように大域的な優先度に基づいて時分割スケジューリングを行うようなことについては論じられていない．

Sun OS 5.0のマルチスレッド機構では，スレッドの優先度はカーネルレベルとユーザレベルで独立しており，優先度に基づくスケジューリングが重要な実時間スレッドに関しては，明示的にユーザスレッドとカーネルスレッドを結合させる必要がある．

カーネルスレッドの管理については，カーネルスレッドのプールを用意しユーザがプールする数を前もって宣言する方式[6)]，余分になったカーネルスレッドを応用プログラムの指示により破棄する方式[8)]，一定時間使用されなかったカーネルスレッドをシステムが自動的に破棄する方式[12)]が提案されている．マイクロプロセス方式では，基本的に，yieldシステムコールにより制御が明け渡された際，set_concurrencyシステムコールで指定したよりも多くの実行可能なカーネルスレッドがあれば，それを破棄する方式を採っている．

また，本研究の特徴としてマイクロプロセスの遠隔手続き呼出し機構が挙げられる．これは，カーネル・ユーザ協調方式のオーバヘッドを回避する目的から設計を行ったものであるが，カーネルスレッドが異なるアドレス空間の手続きを呼び出すための機構として，Machのコンティニュエーションに基づくスタック・ハンドオフ・スケジューリング[11)]やLightweight Remote Procedure Call[5)]と同等の効果をもたらす．カーネル実行コンテクストのハンドオフによりカーネル内でのスレッドの切替えが不要となるため，従来のカーネルスレッドによる遠隔手続き呼出しに比べさらによい性能が得られている．

9. おわりに

マイクロプロセスという名称は，カーネルの支援により横取り可能なユーザスレッドを実現しようとした筆者らの初期の試み[7)]に由来する．

従来のカーネル・ユーザ協調方式では，ユーザスレッドそれぞれにシステムレベルの優先度を割り当て，スケジュールすることができない，また，カーネルスケジューラとユーザスケジューラの協調処理が，むしろシステムの実行効率を下げる場合があるという問題があった．

マイクロプロセス方式では，従来のカーネル・ユーザ協調方式における2階層のスケジューラの協調を推し進め，大域的な優先度に基づく横取り可能なスケジューリングを可能にした．また，遠隔手続き呼出しの実行に際して，カーネルとユーザの協調処理が与える影響を考察し，効率の良い遠隔手続き呼出し機構の提案を行った．さらに，マイクロプロセスをシステムレベルで大域的に識別するための方針を示した．

本論文では，マイクロプロセスの実現機構について，カーネルの役割，ユーザパッケージの役割，カーネルとユーザ間の共有メモリインタフェースに分けて解説した．さらに，筆者らの開発したオペレーティングシステム上に行ったマイクロプロセスの実装とその性能評価について説明した．

参考文献

1) Tevanian, A., Rashid, R., Golub, D., Black, D., Cooper, E. and Young, M.: Mach Threads and the Unix Kernel: The Battle for Control, *Proc. the 1987 USENIX Summer Conference*, pp. 185-197 (July 1987).

2) Sun Microsystems: Lightweight Process Library, *Sun OS Reference Manual, Sun OS Release 4.0* (1988).

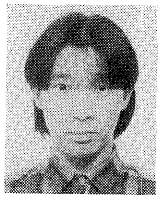
3) Wirth, N. and Gutknecht, J.: The Oberon System, *Software—Practice and Experience*, Vol. 19, No. 9, pp. 857-893 (1989).

4) Leffler, S. J., Mckusick, M. K., Karels, M. J. and Quarterman, J. S.: *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, Reading, Mass. (1989).

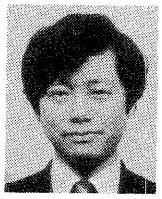
5) Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M.: Lightweight Remote Pro-

cedure Call, *ACM Trans. on Computer Systems,* Vol. 8, No. 11, pp. 33-55 (1990).
6) 新城　靖，清木　康：並列プログラムを対象とした軽量プロセスの実現方式，情報処理学会論文誌，Vol. 33, No. 1, pp. 64-73 (1991).
7) Sinha, P. K., Maekawa, M., Shimizu, K., Jia, X., Ashihara, H., Utsunomiya, N., Park, K.S. and Nakano, H.: The Galaxy Distributed Operating System, *IEEE Computer,* Vol. 24, No. 8, pp. 34-41 (1991).
8) Inohara, S., Kato, K., Narita, A. and Masuda, T.: A Thread Facility Based on User/Kernel Cooperation in the XERO Operating System, *Proc. the Fifteenth IEEE International Computer Software and Applications Conference,* pp. 398-405 (Sep. 1991).
9) Marsh, B., Scott, M., LeBlanc, T. and Markatos, E.: First-Class User-Level Threads, *Proc. the Thirteenth ACM Symposium on Operating Systems Principles,* pp. 110-121 (Oct. 1991).
10) 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. on Computer Systems,* Vol. 10, No. 1, pp. 53-79 (1992).
11) Draves, R., Bershad, B., Rashid, R. and Dean, R.: Using Continuations to Implement Thread Management and Communication in Operating Systems, *Proc. the Thirteenth ACM Symposium on Operating Systems Principles,* pp. 122-151 (Oct. 1991).
12) Stein, D. and Shah, D.: Implementing Lightweight Threads, *Proc. 1992 USENIX Summer Conference,* pp. 1-9 (June 1992).
13) Inohara, S., Kato, K. and Masuda, T.: Unstable Threads: Kernel Interface for Minimizing the Overhead of Thread Switching, *Proc. the Seventh International Parallel Processing Symposium,* pp. 149-155 (Apr. 1993).

(平成6年3月25日受付)
(平成7年1月12日採録)



岡坂　史紀（学生会員）
1970年生．1993年電気通信大学電気通信学部卒業．現在，同大学博士前期課程在学中．オペレーティングシステム，ウィンドウシステムの研究に従事．



清水謙多郎（正会員）
1957年生．1980年東京大学理学部情報科学科卒業．1985年同大学大学院理学系研究科博士課程修了．理学博士．同年東京大学大型計算機センター助手，1986年東京大学理学部助手を経て，1991年より電気通信大学助教授．オペレーティングシステム，並列/分散処理の研究に従事．本学会論文誌編集委員．ACM，IEEE，電子情報通信学会，ソフトウェア科学会各会員．

芦原　　評（正会員）
1964年生．1987年東京大学理学部情報科学科卒業．1992年同大学大学院理学系研究科博士課程修了．理学博士．同年より電気通信大学助手．オペレーティングシステム，並列/分散処理の研究に従事．ACM会員．

亀田　壽夫（正会員）
1942年生．1965年東京大学理学部物理学科卒業．1970年同大学大学院理学系研究科博士課程修了．理学博士．同年東京大学理学部助手．1971年電気通信大学講師．同大学助教授・教授を経て，1992年より筑波大学電子・情報工学系教授．この間，1973～74年IBMワトソン研究所，1974～75年トロント大学において研究に従事．オペレーティングシステム，分散/並列処理，システム性能評価，等の研究教育を行ってきた．本学会編集委員，OS研究会主査等を歴任．ACM，電子情報通信学会，ソフトウェア科学会，日本OR学会，日本応用数理学会各会員．