連続メディア処理向けマイクロカーネルにおける内部排他制御方式

中原雅彦† 岩嵜正明†
竹内理† 中野隆裕†

連続メディア処理においては，スレッドの厳密な周期スケジューリングが必要である．しかし，従来OSのように共有資源の排他制御にロック機構を用いると，スレッドの優先度逆転が発生し，厳密な周期的スケジューリングができない．我々は，連続メディア処理向けマイクロカーネルHiTactixの開発を進めており，その排他制御機構として細粒度プリエンプト制御を開発した．本方式では，クリティカルセクションを複数のプリエンプト禁止区間に細分化する．各プリエンプト禁止区間はプリエンプトされずに実行するが，クリティカルセクション全体としてはプリエンプトを可能とする．この方式により，HiTactixのスレッド・スケジューリング遅延時間が100μ秒以内であることを，HiTactixを用いた試作ビデオサーバシステムの実験により確認した．

A Mutual Exclusion Mechanism of a Micro Kernel for Continuous Media Processing

Masahiko Nakahara,† Masaaki Iwasaki,† Tadashi Takeuchi† and Takahiro Nakano†

It is necessary that an operating system for continuous media processing provides a scheduling algorithm which enables precisely periodic execution of real-time threads. However, if an operating system realizes mutual exclusion mechanism (for shared resources) by lock/unlock, priority inversion will occur. This priority inversion will disturb periodic execution of real-time threads. In this paper, we propose mutual exclusion mechanism by fine-granule preemption to solve this problem. We developed a micro-kernel named HiTactix that employs fine-granule preemption mechanism to achieve high quality continuous media processing. In this method, critical-sections in kernel are divided into non-preemptable code fragments. The execution of each code fragment is never preempted but the execution of the whole critical-section may be preempted. We built an experimental video server system using HiTactix and measured its scheduling performance. We confirmed that scheduling jitter of periodic threads never exceeds 100 microseconds in that system.

1. はじめに

インターネットの爆発的な普及を背景に，VOD（Video On Demand）等の，ネットワークを利用した連続メディアを処理するアプリケーションへの需要が高くなっている．

一般に，連続メディア処理を行うスレッドは，厳密に一定の周期で動作することが要求される．そのため，連続メディア処理の実現には，カーネルがスレッドの周期駆動を保証するスケジューラを備える必要がある．連続メディア処理用に開発されたRialto[9]，Nemesis[10]，RT-Mach[11]等のカーネルでは，そのための独自スケジューラが備えられている．

一方，現在研究・開発されている連続メディア処理用カーネルの多くは，マルチスレッド機構をサポートしている．この場合，カーネル内部における共有資源の排他制御が必要になる．一般的には，この排他制御にロックを使用している．しかし，ロックを使用してカーネル内共有資源の排他制御を行うと，優先度逆転（Priority Inversion）問題が発生する．優先度逆転問題とは，優先度の低いスレッドが共有資源をロックしたまま休眠したときに，優先度の高いスレッドが共有資源のロックを獲得できず，実行不能に陥ることをいう．優先度逆転が発生すると，スレッドのスケジューリング遅延時間が大きくなり，スレッドの厳密な周期駆

† 株式会社日立製作所システム開発研究所
Systems Development Laboratory, Hitachi Ltd.

動が不可能になる．リアルタイム・スケジューリング・アルゴリズムである Priority Ceiling Protocol[1] は，スレッドが自分より優先度の低いスレッドにブロックされる回数がたかだか1回となるアルゴリズムにより，この問題を回避している．この方式では，スレッドの最大スケジューリング遅延時間は共有資源をロックしている時間となる．このとき，共有資源をロックしている時間が数十マイクロ秒程度なら連続メディア処理のスケジューリング遅延時間として問題とはならないが，数百マイクロ秒～数ミリ秒に及ぶ場合はスレッドの厳密な周期駆動を維持できなくなる．すなわち，周期駆動を実現するには，ロック時間の短縮を図ることが第1の課題である．また，ロックしている時間が短い場合でも，優先度の高いスレッドがブロックされた場合，ロックを保持しているスレッドを優先して実行させるためにコンテキスト・スイッチが発生し，処理オーバヘッドが大きい．したがって，ロック制御，および，それにともなうコンテキスト・スイッチによる処理オーバヘッドを低減することが周期駆動を実現するための第2の課題である．

我々は，連続メディア処理向けマイクロカーネル HiTactix の開発を進めている[2)~6)]．HiTactix では，前述した2つの課題を解決するために，カーネル内共有資源の排他制御にはいっさいロックを使用せず，細粒度プリエンプト制御方式を採用することにより，スケジューリング遅延時間 100 μ 秒以下を実現した．本稿では，次章でこの細粒度プリエンプト制御について述べ，3 章では HiTactix メモリ管理を例として，細粒度プリエンプト制御の実装について述べる．4 章において実アプリケーションを用いた細粒度プリエンプト制御の評価について述べる．また，5 章では関連研究について述べる．

2. 細粒度プリエンプト制御

共有資源の排他制御にともなう優先度逆転問題を回避して周期駆動を実現するため，HiTactix ではカーネル内部の排他制御区間を細分化することにより，各区間の実行時間が一定時間以内に収まるように設計した．これにより，カーネル内資源解放待ち時間が一定時間以内であることが保証可能となり，周期駆動を行うスレッド（以下周期スレッド）のスケジューリング遅延時間を一定時間以内に収めることを保証できる．ただし，排他制御に Priority Ceiling Protocol によるロック/アンロック機構を用いると，先にも述べたように，資源解放待ちが発生した際のロック解放に必要なコンテキスト・スイッチの処理オーバヘッドが大きく，排他制御区間の細分化の効果を減少させてしまう．

このため，HiTactix では排他制御にロックを使用せず，各排他制御区間をプリエンプト禁止状態で実行する方式を採用した．この方式を細粒度プリエンプト制御と呼ぶ．この方式では，タイマ割り込み等が発生したときに実行中のスレッドがプリエンプト禁止状態であった場合，スケジューラは実行中のスレッドがプリエンプト可能状態になるまでコンテキスト・スイッチを遅延させる．したがって，プリエンプト可能となるまでのスケジューリング遅延は発生するが，Priority Ceiling Protocol のような資源解放に必要なコンテキスト・スイッチの処理オーバヘッドはなくなる．

スケジューリング遅延時間を最小に抑えるためには，最長プリエンプト禁止時間は短いほど良い．しかし，スケジューリング遅延が発生したときに，プリエンプト禁止解除後に必要となるコンテキスト・スイッチの処理オーバヘッドを考慮すると，コンテキスト・スイッチの時間より短い時間を最長プリエンプト禁止時間に指定することは現実的ではない．一般的にはコンテキスト・スイッチに数十マイクロ秒を必要とする．たとえば，HiTactix では 200 MHz の Pentium☆において☆☆，最悪ケースのコンテキスト・スイッチ・オーバヘッドは約 29 μ 秒である[8)]．さらに，プリエンプト禁止，および，その解除の制御に要する処理オーバヘッド☆☆☆も考慮すると，最長プリエンプト禁止時間を数十マイクロ秒の後半～100 μ 秒よりも小さく設定することに意味がない．

上記の理由により，HiTactix では最長プリエンプト禁止時間が 100 μ 秒を超えないように設計した．本章では，この細粒度プリエンプト制御について説明する．

なお，細粒度プリエンプト制御はシングルプロセッサ・システムを前提としている．

2.1 カーネル・インタフェースのクラス分類

HiTactix では，カーネル・インタフェースをリアルタイム・クラス（以下 RT クラス）と非リアルタイム・クラス（以下 NRT クラス）の2種類に分類している．RT クラスは，プロセス識別子の取得等のように，実行時間が予測可能なカーネル・インタフェースである．一方，NRT クラスは，プロセス生成等のように，実行時間に数ミリ秒を要し，通常は実行時間の予測が困難なカーネル・インタフェースである．

RT クラスのカーネル・インタフェース内部処理（以

☆ Pentium は米国 Intel Corporation の登録商標である．
☆☆ HiTactix の性能設計に使用する基準ハードウェアである．詳しくは 3.1 節で述べる．
☆☆☆ 4.1 節にて評価する．

下内部処理）については，前述したように排他制御区間の細分化により，一定の最長プリエンプト禁止時間を超える資源獲得処理が発生しないように設計した．さらに，周期スレッドからは，このRTクラスのカーネル・インタフェースのみを呼び出し可能とし，NRTクラスのカーネル・インタフェースは呼び出さないというプログラミング規約を設けることにより，数百マイクロ秒以上を要する資源確保によってスレッドの周期駆動性が失われることを回避している．周期スレッドの実行において，NRTクラスのカーネル・インタフェースを使用する資源が必要な場合は，次のようにして資源を確保する．

HiTactixでは，プログラムの実行に必要な資源を割り当てる対象としてプロセスが存在し，プロセスに割り当てられた資源を使用してプログラムを実行する単位としてスレッドが存在する．周期スレッドを使ってプログラムを実行する場合，基本的には次の手順にて周期スレッドを実行する．

(1) プロセスを生成する．
(2) プログラムの実行に必要な資源（メモリ等）を確保し，生成したプロセスに割り当てる．
(3) スレッドを生成する☆．
(4) 生成したスレッドの周期駆動開始をカーネルに要求する☆☆（このとき，駆動周期と割当てCPU時間を指定する）．

このように，NRTクラスのカーネル・インタフェースについては，その資源予約を周期駆動開始前に実施することで対応可能であり，上記プログラミング規約はアプリケーションを作成するうえで，それほど大きな制約とはならない．

2.2 プリエンプト可能/アボート禁止区間

ここでは，カーネルの内部処理をプリエンプト可能にするためのアボート禁止区間について説明する．

カーネルの内部処理においては，RTクラス，NRTクラス双方に資源へのアクセス競合が発生する可能性がある．このアクセス競合が発生する部分（以下クリティカルセクション）については，RTクラス，NRTクラスとも，その内部処理をプリエンプト禁止で実行する．クリティカルセクションが最長プリエンプト禁止時間を超える場合は，複数のプリエンプト禁止区間に分割する．各プリエンプト禁止区間内では，そのスレッドの処理がプリエンプトされないことを保証する．クリティカルセクションの処理全体では，その処理を実行中のスレッドをプリエンプトすることが可能である．

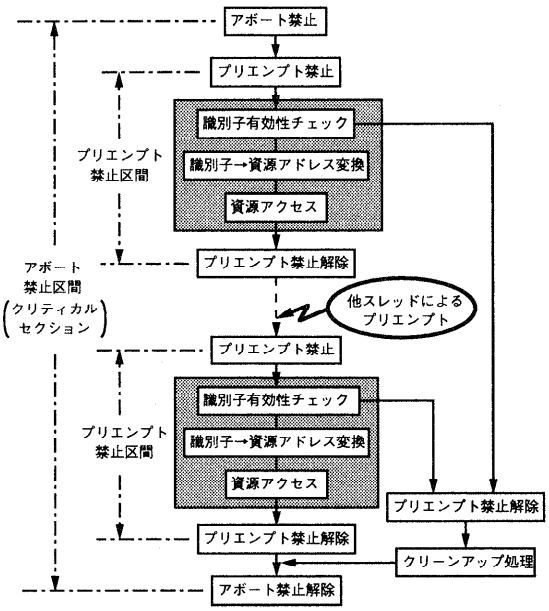


図1 細粒度プリエンプト制御の流れ
Fig. 1 Flow of fine-grain preemption.

しかし，一般にプリエンプト可能状態においては，スレッドのアボート（強制終了）が可能である．クリティカルセクション実行中のスレッドが実行途中でアボートされると，カーネル内部状態の一貫性が損なわれる．

プリエンプト可能なクリティカルセクション内においてスレッドがアボートされることを防ぐため，HiTactixの各スレッドはカーネル内処理開始前に“アボート禁止”を宣言する．カーネル内処理終了時には“アボート禁止解除”を宣言し，スレッドの状態を元に戻す（**図1**を参照）．

このように，HiTactixでは処理時間が最長プリエンプト禁止時間を超えるクリティカルセクションを“プリエンプト可能/アボート禁止”状態で実行する．これらの要素機能と，次節で述べる機能を組み合わせることにより，細粒度プリエンプト制御が可能となる．

2.3 資源アクセス管理

HiTactixでは，資源の不正な消去を防止するため，資源を生成したプロセスにのみ消去の権限を与えている．このため，アプリケーションが回復不能な例外

☆ この時点では，周期スレッド，通常スレッド（非周期スレッド）の区別はない．ステップ4において周期駆動開始をカーネルに要求した際，周期スレッドとなる．また，スレッドの生成は同一プロセス内のスレッドからも，他プロセスのスレッドからも可能である．

☆☆ HiTactixでは，周期スレッド，通常スレッドとも，カーネルに明示的に実行開始を要求しない限りスレッドにCPUを割り当てない．

を発生させた等の理由によりプロセスが消去される場合，そのプロセスが生成した資源も解放される．この解放される資源を他のプロセスが参照している場合，解放処理において，その資源を参照しているすべてのプロセスに資源解放を通知する必要がある．このとき，カーネル内資源へのアクセスの一貫性を保つために，資源解放の通知が完了するまでシステム内で動作している全スレッドを停止させると，スレッドの周期駆動が阻害される問題が発生する．

HiTactix では，資源解放を通知する代わりに，各資源にユニークな識別子を付与し，この識別子の有効性をチェックして資源の消去を検知する方法を採用し，上記問題を解決している．資源へのアクセスは，識別子による間接的なアクセスとなる．スレッドは，各プリエンプト禁止区間内において最初に共有資源へアクセスする際に，変換関数を使用して識別子を資源のアドレスに変換する．変換された資源アドレスは，そのプリエンプト禁止区間内でのみ使用できる．

共有資源を消去したスレッドは，同時にその共有資源に対する識別子の変換エントリも無効にする．仮に共有資源の状態が 2 つのプリエンプト禁止区間の間，すなわちプリエンプト可能区間を通る間に変更されていた場合，識別子から資源アドレスへの変換に失敗し，資源にアクセスしようとしたスレッドは資源へのアクセスが無効であることを検知することができる（図 1 を参照）．

2.2 節 において述べたプリエンプト可能/アボート禁止機能と本節の資源アクセス管理を組み合わせた細粒度プリエンプト制御の処理の流れを図 1 を使って説明する．2.2 節 でも述べたように，クリティカルセクションを処理するスレッドは，処理開始前にアボート禁止を宣言する．共有資源にアクセスする際は，共有資源アクセス開始前にプリエンプト禁止を宣言し，他スレッドにプリエンプトされないようにする．次に，共有資源の識別子が有効であるか否かをチェックし，有効ならば資源アドレスに変換し，共有資源にアクセスする．識別子が無効である場合はクリーンアップ処理に入る（クリーンアップ処理に入る前にプリエンプト禁止は解除する）．

クリティカルセクションが最長プリエンプト禁止時間を超える場合は，いったんプリエンプト禁止を解除し，他スレッドへのプリエンプトを可能にする．再度，共有資源にアクセスが必要な場合は，プリエンプト禁止を宣言し，共有資源の識別子を使って資源の有効性を再度チェックする．クリティカルセクションが終了したら，最後にアボート禁止を解除する．クリティカルセクションが最長プリエンプト禁止時間を超えるかどうかの判断方法については，3.3 節 で述べる．

以上の制御により，クリティカルセクションをプリエンプト可能とし，かつカーネル内部状態の一貫性を保つ細粒度プリエンプト制御を実現している．

一般の OS，たとえば UNIX では，共有資源に参照カウンタを設け，参照カウントがゼロになった際に資源を解放する手法を採用している．UNIX では，親プロセスが子プロセスを生成する際に資源が継承されることによって共有され，親プロセスが消滅する際に子プロセスも消滅し，確実に資源が解放されるためである[12]．

HiTactix では，子プロセスが生成される際，カーネルが動作するためのメモリ等一部の資源を除き，資源の継承は行わない．また，親プロセスと子プロセスの間に依存関係はなく，親プロセスが消滅しても子プロセスは生存し続ける仕様としている．HiTactix はマイクロカーネルと OS サーバにより OS を構築しており，OS サーバ設計の際に制約を与えないためである．したがって，子プロセスが親プロセスの資源を共有する場合は，明示的に資源の共有，および，解放を行う必要がある．参照カウンタによる共有アクセス管理を行う場合，プログラミング・ミス等を原因とする不当な資源の解放等から参照カウンタを保護するためには，各資源ごとに参照プロセスを管理する必要があり，管理データ量が多くなる．また，子プロセスが資源の解放を忘れた場合，その資源が永久に残ることになる．これらは，カーネルのデバッグを困難にする要因となる．

このような問題を避けるため，HiTactix では，資源を生成したプロセスにのみ消去の権限を与え，他のプロセスは参照のみ行うことができるようにしている．

2.4 逐次化スレッド

NRT クラスの内部処理では，共有資源へのアクセス時間が数百マイクロ秒以上を要する場合がある．周期スレッドのスケジューリング遅延を一定時間以内に抑えるためには，このような NRT クラスの共有資源アクセス処理も複数のプリエンプト禁止区間に分割し，プリエンプト可能にする必要がある．しかし，NRT クラスのカーネル・インタフェース呼び出しは複数のスレッドから同時に発生する可能性がある．NRT クラスの内部処理をプリエンプト可能にすると，共有資源のアクセス競合が発生し，一貫性を保つことが困難となる．この問題は，一般にロック機構や逐次化機構を使って解決できる．HiTactix においても，以下に述べる逐次化技術の一種である逐次化スレッドにより問

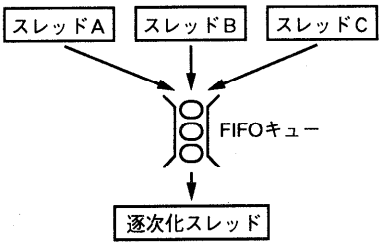


図 2 逐次化スレッド
Fig. 2 Serializing thread.

題を解決している．

逐次化スレッドは，カーネル内の共有資源に唯一アクセス可能なスレッドである．NRT クラスの内部処理は，**図 2** に示すように FIFO キューを介して逐次化スレッドに送られる．この結果，NRT クラスの内部処理は逐次化され，共有資源へのアクセス競合が発生しないため，NRT クラスの内部処理をプリエンプト可能にすることができる．すなわち，常時優先度の高い周期スレッドの優先処理が可能となり，周期スレッドの周期駆動が実現できる．

3. メモリ管理における細粒度プリエンプト制御の実装

本章では，メモリ管理を例として，HiTactix における細粒度プリエンプト制御の実装について説明する．

HiTactix では，アプリケーション間のメモリ保護，およびアドレス空間内のフラグメント化を防止するため，多重アドレス空間構成を導入している．仮想アドレス空間内は「領域」を単位として管理し，物理メモリの割当ては「物理ページ集合」を単位として管理している．HiTactix では，多重アドレス空間構成における性能低下を回避するため，領域および物理ページ集合を仮想アドレス空間間で共有できる機能を設けている[4]．

一方，HiTactix ではマルチスレッド機構を採用しているため，複数のスレッドから同時にプロセス生成要求が発生する可能性がある．プロセス生成では，アプリケーションをロードするためのメモリ確保等，メモリ管理インタフェースの処理を多用する．したがって，メモリ管理インタフェースでは共有資源への処理競合が発生する可能性が高い．メモリ管理における細粒度プリエンプト制御の実装が，スレッドの周期スケジューリングのためには重要となる．

そこで，HiTactix メモリ管理における細粒度プリエンプト制御の適用方法を 2 章の項目に従って述べる．

表 1 設計基準マシン
Table 1 Design standard machine.

項目	性能
CPU	Pentium 200 MHz
2 次キャッシュ	512 KB
チップセット	Intel 430 HX
メモリ	64 MB

表 2 メモリ管理インタフェース
Table 2 Memory management interface.

関数名	クラス	機能
vm_allocate_region	RT	領域の生成
vm_share_region	NRT	領域の共有
vm_deallocate_region	NRT	領域の解放
vm_destroy_region	NRT	領域の強制消去
vm_get_region_info	RT	領域情報の要求
vm_set_region_info	NRT	領域情報の設定
vm_get_phys_page_set_list	RT	マップ情報の要求
vm_get_region_holder_list	RT	参照情報の要求
pm_allocate_phys_page_set	RT	PPS の生成
pm_share_phys_page_set	NRT	PPS の共有
pm_deallocate_phys_page_set	NRT	PPS の解放
pm_destroy_phys_page_set	NRT	PPS の強制消去
pm_map_phys_page_set	NRT	PPS のマップ
pm_unmap_phys_page_set	NRT	PPS のアンマップ
pm_get_phys_page_set_info	RT	PPS 情報の要求
pm_set_phys_page_set_info	NRT	PPS 情報の設定

注）PPS は物理ページ集合（Physical Page Set）の略

3.1 前提となるハードウェア条件

カーネル・インタフェースのクラス分類やプリエンプト禁止区間の設計は実際のソフトウェア処理時間によって決定されるが，ソフトウェアの処理時間は CPU 動作周波数等の要因で変化する．したがって，これらの決定に際しては，設計基準となるハードウェアが必要である．HiTactix では，**表 1** のマシンを設計基準マシンとし，カーネル・インタフェースのクラス分類，プリエンプト禁止区間等を決定している．

3.2 メモリ管理インタフェースのクラス分類

メモリ管理インタフェースと，表 1 の設計基準マシンを基にしたクラス分類を**表 2** に示す．表 2 中の RT は RT クラス，NRT は NRT クラスのインタフェースであることを表している．

メモリ管理インタフェースの内部処理において，排他制御を必要とするのは内部管理テーブル更新時である．メモリ管理の内部管理テーブル更新処理，特にページテーブルの更新に要する時間は処理する物理メモリ量によって異なり，処理全体をプリエンプト禁止にすることはできない．さらに，HiTactix ではアドレス空間間で領域および物理ページ集合が共有可能であるため，同一の領域/物理ページ集合に対して異なる処理が並行して発生する可能性がある．ページテーブル

表 3 アボート/プリエンプト制御用関数
Table 3 Abort/fine-grain preemption control functions.

関数名	機能
xthread_abort_disable	自スレッドをアボート禁止にする
xthread_abort_enable	自スレッドのアボート禁止状態を解除する
xthread_preempt_disable	自スレッドをプリエンプト禁止にする
xthread_preempt_enable	自スレッドのプリエンプト禁止状態を解除する

更新処理をプリエンプト可能状態で実行すると，ページテーブルの一貫性を保つことは困難である．このため，領域および物理ページ集合の生成や，管理情報の取得等，ページテーブル更新を行わないインタフェースについてはRTクラスに，物理ページ集合のマップや領域/物理ページ集合の消去等，ページテーブルの更新をともなうインタフェースをNRTクラスとして分類している．

3.3 プリエンプト可能/アボート禁止の制御

メモリ管理インタフェースは，RTクラス，NRTクラスともにその開始から終了までをアボート禁止状態で処理する．また，RTクラス，NRTクラスとも，表1の設計基準マシン上での実行時間を基に，その内部処理を複数のプリエンプト禁止区間に分割しており，他スレッドによるプリエンプトが可能である．

アボート禁止/アボート禁止解除，および，プリエンプト禁止/プリエンプト禁止解除の制御には，**表3**に示す関数を使用する．プリエンプト禁止区間が最長プリエンプト禁止時間以内に収まっているか否かは，このプリエンプト禁止/プリエンプト禁止解除関数により確認可能である．プリエンプト禁止関数が呼び出されてプリエンプト禁止状態に入ったときから，プリエンプト禁止解除関数が呼び出されてプリエンプト禁止が解除されるまでの時間をカーネル内で計測し，その時間が最長プリエンプト禁止時間を超えている場合は警告を出す．時間の計測には，表1の設計基準マシンのCPUクロックカウンタを用いる．

また，これらの関数は**図3**のようにネストして呼び出すことを可能としている．アボート禁止，プリエンプト禁止のネスト数はカーネル内で管理しており，アボート禁止，プリエンプト禁止関数が呼び出された際にインクリメントし，ネスト数を関数の引数で指定された場所に返す．返されたネスト数は，対になるアボート禁止解除，プリエンプト禁止解除関数を呼び出す際の引数として使用する．アボート禁止解除，プリエンプト禁止解除関数は，引数として渡されたネスト数と，カーネル内部で管理しているネスト数を比較し，値が一致しない場合はエラーを返す．この機能により，アボート禁止解除，プリエンプト禁止解除欠落等のプログラミング・ミスを検出可能にしている．たとえば，図3の例において，(a)のプリエンプト禁止関数のcount1には“1”が，(b)ではcount2に“2”が返される．ここで，プログラマが(c)のプリエンプト禁止解除関数を入れ忘れたとする．(d)のプリエンプト禁止解除関数において，引数にはcount1の値，すなわち“1”を渡すが，カーネル内部で管理しているプリエンプト禁止のネスト数は“2”であるため，(d)のプリエンプト禁止解除関数はエラーを返す．

```
xthread_preempt_disable(&count1);  ……（a）
        :
xthread_preempt_disable(&count2);  ……（b）
        :
xthread_preempt_enable(count2);    ……（c）
        :
xthread_preempt_enable(count1);    ……（d）
```

図3 プリエンプト制御関数のネスト
Fig. 3 Nesting of fine-grain preemption.

3.4 資源アクセス管理

メモリ管理において管理対象となる資源は，仮想メモリ，物理メモリの管理単位である領域および物理ページ集合である．領域および物理ページ集合は，それらを生成するごとに資源管理テーブルに登録し，資源識別子を割り当てる．資源管理テーブルは**図4**のように構成されており，資源が登録されるごとに1個のエントリを割り当てる．割り当てたエントリには，その資源アドレスを格納し，さらに世代カウンタを更新する．同時にフラグをエントリ使用状態に変更する．登録された資源に対しては，割り当てた資源管理テーブル内エントリのインデクス（これを資源識別番号と呼ぶ）と世代カウンタとから構成される資源識別子を割り当てる（図4参照）．世代カウンタは資源管理テーブルのエントリが再利用されるごとに更新されるので，資源管理テーブルの同じエントリが再利用されても，資源識別子には異なった値が割り当てられる．したがって，この資源識別子を使用することにより，プリエンプト可能区間において他スレッドにより資源が消去されたことを検出することができる．

3.5 メモリ管理スレッド

NRTクラスのメモリ管理インタフェースは，専用の逐次化スレッド（以下メモリ管理スレッド）が処理する．メモリ管理スレッドは通常スレッドであり，他スレッドからプリエンプト可能である．

NRTクラスおよびRTクラスのメモリ管理インタフェース処理概要を**図5**に示す．アプリケーションが

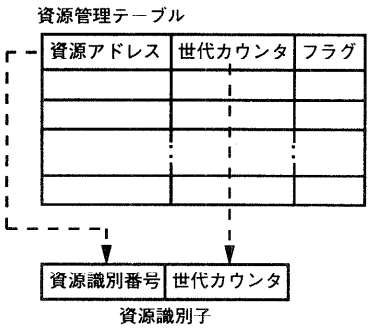


図 4 資源管理テーブルと資源識別子
Fig. 4 Resource management table and resource identifier.

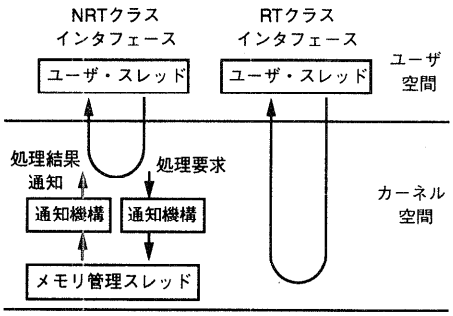


図 5 メモリ管理インタフェースの処理
Fig. 5 Memory management interface processing.

NRT クラスのメモリ管理インタフェースを呼び出した場合，ユーザ・スレッドがカーネル空間にマイグレートし，メモリ管理スレッドに処理要求を通知する．メモリ管理スレッドは，処理要求を受理すると要求された処理を行い，結果を要求元に通知する．ユーザ・スレッドはメモリ管理スレッドから処理完了が通知されるまで待機し，結果を受理するとユーザ空間にリターンする．周期スレッドが NRT クラスのメモリ管理インタフェースを呼び出した場合は，メモリ管理スレッドに処理は渡らず，ただちにエラーリターンする．

RT クラスのメモリ管理インタフェースは，ユーザ・スレッドがカーネル空間にマイグレートし，カーネルのコードを直接実行する．

4. 評 価

ここでは，細粒度プリエンプト制御の処理オーバヘッド，および，有効性について評価する．

4.1 細粒度プリエンプト制御のオーバヘッド

本節では，細粒度プリエンプト制御の処理オーバヘッドについて検証する．

HiTactix に Priority Ceiling Protocol を適用したと仮定した場合，周期スレッドがブロックされたときのスレッド切替えの処理オーバヘッドは，**表 4** に示す HiTactix のコンテキスト・スイッチ・オーバヘッドの数値から，最小の場合でも 36.4 μ 秒と推定される．一方，細粒度プリエンプト制御における各制御関数（表 3 参照）の処理オーバヘッドは**表 5** のとおりであり，プリエンプト禁止区間が 1 つの場合，最大でも 3.3 μ 秒である．実際に HiTactix 上でビデオサーバソフト Miles[7] を実行した場合では，表 5 の各関数とも平均 0.3〜0.4 μ 秒で実行しており，プリエンプト禁止区間が 1 つの場合の平均処理オーバヘッドは約 1.4 μ 秒となる．この値は，Priority Ceiling Protocol において周期スレッドのブロックが発生した場合の 4%程度である．

表 4 コンテキスト・スイッチ・オーバヘッド
Table 4 Context switch overheads.

最小	最大
18.2	29.2

注）単位は μ 秒．

表 5 アボート/プリエンプト制御関数オーバヘッド
Table 5 Abort/fine-grain preemption overheads.

関数名	実行時間	
	最小	最大
xthread_abort_disable	0.2	0.6
xthread_abort_enable	0.2	1.1
xthread_preempt_disable	0.2	0.6
xthread_preempt_enable	0.2	1.0

注）単位は μ 秒．

また，プリエンプト禁止区間を細分化すること，すなわち，プリエンプト制御関数を繰り返し呼び出すことによる処理オーバヘッド増加についても，表 3 のメモリ管理インタフェースの中で最もプリエンプト禁止区間の分割数が多い pm_unmap_phys_page_set において，分割しない場合に比べて平均 2.7 μ 秒，最大 5.1 μ 秒のオーバヘッドである．pm_unmap_phys_page_set の実行時間は最小で約 130 μ 秒であり，細分化によるオーバヘッドは実行時間全体の 4%未満である．

細粒度プリエンプト制御は，既存の排他制御方式に比べて処理オーバヘッドは小さいといえる．

4.2 スケジューリング遅延時間

本節では，優先度逆転問題に対する細粒度プリエンプト制御の有効性を，実アプリケーションを使用した測定により評価する．

測定環境を**図 6** に，使用したマシンの仕様を**表 6** に示す．ビデオサーバは 8 台のクライアントと 100 Mbps Ethernet を介して接続されている．ビデオサーバ上では OS として HiTactix が動作し，その上でビデオサーバソフト Miles[7] が稼動している．クライアント

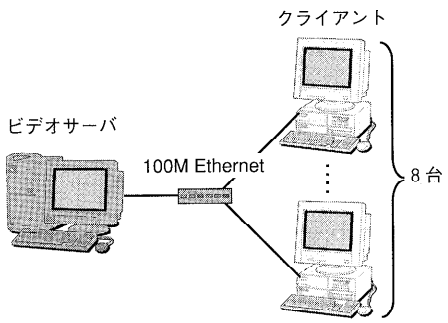


図6 測定環境
Fig. 6 Evaluation environment.

表6 測定マシン仕様
Table 6 Machine specification of the measurement environment.

項目	ビデオサーバ	クライアント
CPU	Pentium	Pentium
動作周波数	200 MHz	166 MHz
1次キャッシュ	16 KB	16 KB
2次キャッシュ	512 KB	256 KB
メモリサイズ	64 MB	32 MB
チップセット	Intel 430HX	Intel 430HX
OS	HiTactix	Windows®95
アプリケーション	ビデオサーバ	ビデオ再生ソフト
備考		MPEG1 はハードウェアデコード

上ではOSとしてWindows®95☆が動作し，その上でビデオ再生ソフトが稼動する．ビデオサーバは1タイトルあたり1〜20分のMPEG1ビデオデータを14タイトル保有し，クライアントの要求に従ってビデオデータを送信する．クライアントは，ランダムにタイトルを選択し，ビデオサーバに送信を要求する．クライアントは，1つのタイトル再生が終了するごとに同様の処理を繰り返す．

この環境において，ビデオサーバとクライアント8台の間で10分間ビデオ配送運転をさせたときの最長プリエンプト禁止時間，および，スケジューリング遅延時間を測定した．ここでいう遅延時間とは，**図7**に示すように，プリエンプト禁止により発生する周期スケジューリング遅延時間ΔTのことを指す（Tはスレッドの起動周期，Lは1周期あたりのスレッド実行時間を表す）．

測定は次の2種類のケースで行った．

☆ Windows®は米国Microsoft Corporationの米国およびその他の国における商標または登録商標である．また，Windows®95の正式名は，Microsoft® Windows® Operating System 95である．

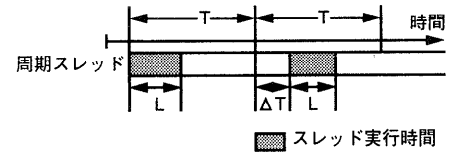


図7 スケジューリング遅延時間
Fig. 7 Scheduling jitter.

表7 測定結果
Table 7 Measurement result.

	CPU利用率（%）	プリエンプト禁止時間			スケジューリング遅延時間		
		最小	最大	平均	最小	最大	平均
(a)	7.9	1	66	4	1	50	1
(b)	100.0	1	85	6	1	55	2

注）(a) はケース1，(b) はケース2の測定結果，単位は μ 秒．

(a) ケース1
ビデオサーバ上においてMilesのみ動作している場合

(b) ケース2
ビデオサーバ上においてMilesと並行してプロセスの生成・消去を反復実行する負荷プログラムが動作する場合

ケース1では，単純に連続メディア処理を行うアプリケーションをHiTactix上で実行させた場合のプリエンプト禁止時間とスケジューリング遅延時間を測定している．ケース2は，連続メディア処理を行うアプリケーションと並行して，他のアプリケーションが頻繁に実行/終了される場合を模して，その場合のプリエンプト禁止時間とスケジューリング遅延時間を測定している．

測定結果を**表7**に示す．ケース1では，Milesのみの実行であるため，CPU利用率が低く，スケジューリング遅延は発生しにくい．実際，最大プリエンプト禁止時間66 μ 秒，最大スケジューリング遅延時間50 μ 秒であり，細粒度プリエンプト制御の設計目標値を満たしている．

一方，ケース2ではMilesと並行してプロセスの生成・消去を反復するアプリケーションが走行しているため，CPU利用率は100%となっている．このようなCPU高負荷環境においても，最大プリエンプト禁止時間85 μ 秒，最大スケジューリング遅延時間55 μ 秒であった．これは，プロセス生成・消去のような実行時間に数ミリ秒を要するカーネル・インタフェース実行中でも，厳密な周期スケジューリングが実行できていることを表している．細粒度プリエンプト制御が，周期スケジューリングに必要不可欠な，優先度逆転問

題の解決方法として有効であることが分かる．

5. 関連研究

現在までに，Rialto[9]，Nemesis[10]，RT-Mach[11]をはじめとする数多くの連続メディア向きオペレーティングシステムが開発されている．

Rialto，Nemesis，RT-Machは共有資源の排他制御にロックを使用している．ロック使用にともなって発生する優先度逆転に対しては，Priority Ceiling Protocol[1]と同様の方式により回避している．この方式は，最初にも記したように，スレッドが自分より優先度の低いスレッドにブロックされる回数はたかだか1回というアルゴリズムにより，最大スケジューリング遅延時間を保証しようとしている．また，細粒度プリエンプト制御がシングルプロセッサ・システムを前提としているのに対して，前記方式はマルチプロセッサ・システムでもそのまま適用可能という利点がある．しかし，優先度の高いスレッドがブロックされた場合に，そのスレッドをブロックしたスレッドを優先して実行させるために2回のコンテキスト・スイッチを必要とし，HiTactixの細粒度プリエンプト制御と比較して，処理オーバヘッドが大きい．この点は4.1節で述べたとおりである．

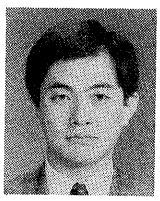
6. おわりに

本稿では，排他制御における優先度逆転問題を解決する方法として，細粒度プリエンプト制御の概要と評価について述べた．この方式により，資源の排他制御にともなう優先度逆転問題を回避し，スレッドの厳密な周期スケジューリングを可能とした．さらに，細粒度プリエンプト制御によるスケジューリング遅延時間が100μ秒未満で実現していることを，実アプリケーションを用いた性能測定により確認し，その有効性を示した．

参考文献

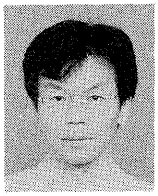
1) Sha, L., et al.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Trans. Comput.*, Vol.39, No.9, pp.1175–1185 (1990).
2) 岩嵜ほか：連続メディア処理向きマイクロカーネル HiTactixの設計と評価，情報処理学会コンピュータシステム・シンポジウム論文集，pp.99–104 (1996).
3) 竹内ほか：連続メディア処理向きOSの周期駆動保証機構の設計と実装，情報処理学会論文誌，Vol.40, No.3, pp.1204–1215 (1999).
4) 中原ほか：連続メディア処理向きマイクロカーネルの開発 (3)—メモリ管理の開発，第53回情報処理学会全国大会論文集，pp.145–146 (1996).
5) Iwasaki, M., et al.: An Experiment of Isochronous Video Data Transfer on Networks, *Worldwide Computing and Its Applications*, pp.334–349 (1997).
6) Iwasaki, M., et al.: Isochronous Scheduling and its Application to Traffic Control, *19th IEEE Real-Time Systems Symposium*, pp.14–25 (1998)
7) 中村ほか：連続メディア処理向きマイクロカーネルの開発 (5)—ビデオサーバ動画配送高速化方式の開発，第53回情報処理学会全国大会論文集，pp.149–150 (1996).
8) 川田ほか：連続メディア処理向けカーネルの性能評価，第57回情報処理学会全国大会論文集，pp.52–53 (1998).
9) Jones, M.B. and Rosu, D.R.M.-C: CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, *16th ACM Symposium on Operating Systems Principles*, pp.198–211 (1997).
10) Leslie, I., et al.: The Design and Implementation of Operating System to Support Distributed Multimedia Applications, *IEEE Journal on Selected Areas in Communications*, Vol.14, No.7, pp.1280–1297 (1996).
11) Mercer, C.W., Savage, S. and Tokuda, H.: Processor Capacity Reserves: Operating System Support for Multimedia Applications, *Proc. International Conference on Multimedia Computing and Systems*, pp.90–99 (1994).
12) Bach, M. J.: *The Design of the UNIX Operating System*, Prentice-Hell (1986).

(平成10年12月1日受付)
(平成11年 4 月1日採録)



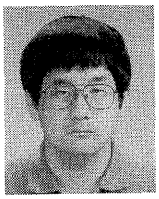
中原 雅彦（正会員）

昭和40年生．昭和63年東京農工大学工学部数理情報工学科卒業．平成2年同大学大学院工学研究科修士課程修了．同年（株）日立製作所システム開発研究所入社．入社以来，ワークステーションの性能評価，並列計算機用オペレーティング・システム，連続メディア処理向きマイクロカーネル等の研究・開発に従事．



岩嵜　正明（正会員）

昭和33年生．昭和56年九州工業大学工学部電子工学科卒業．昭和58年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．同年（株）日立製作所中央研究所入所，平成5年より同社システム開発研究所勤務．並列推論マシンPIM，メインフレームSMPシステム，並列計算機SR2201のOS研究開発を経て，HiTactixの研究を開始，現在に至る．



竹内　　理（正会員）

昭和44年生．平成4年東京大学理学部情報科学科卒業．平成6年同大学大学院理学系研究科情報科学専攻修士課程修了．同年（株）日立製作所システム開発研究所入社．連続メディア処理向きマイクロカーネルの研究，特にリアルタイムスケジューリング方式，リアルタイム通信方式，ヘテロジニアスOSアーキテクチャに関する研究に従事．

中野　隆裕（正会員）

昭和44年生．平成5年電気通信大学電気通信学部情報工学科卒業．平成7年同大学大学院電気通信学研究科情報工学専攻修士課程修了．同年（株）日立製作所システム開発研究所入社．連続メディア処理向きマイクロカーネルの研究，特に入出力方式やリアルタイム通信方式に関する研究・開発に従事．