単一2次元アドレス空間を提供する拡張可能なマイクロカーネルの開発

森永智之† 早川栄一†
並木美太郎† 高橋延匡†

マイクロカーネルは対象に応じた資源管理を提供できるので，多様なデータを扱うパターンデータ処理に有効である．しかし，既存のマイクロカーネルは互いにリンクされたデータを扱うことが難しいという問題があった．また，ユーザが保護のポリシーを決定できず，信頼性と性能とのトレードオフを図ることが困難であった．我々は，マイクロカーネルで単一2次元アドレス空間を提供することによって，データ間のリンクをポインタで表現し，保護の問題を解決する．単一2次元アドレス空間では，線形なアドレス空間を持つセグメントの集合からなる2次元のアドレス空間を全タスクで共有する．また，コンテキスト切替えを減少させるthrowと呼ぶ構成プリミティブを提供し，ダイナミックリンクによって資源管理部を拡張することで高速で柔軟なシステム構成を可能にする．このマイクロカーネルでは，ユーザが保護のポリシーを設定できる保護空間と呼ぶ保護機構を提供することで，信頼性と性能とのトレードオフを図ることを可能にする．さらに，我々はマイクロカーネルアーキテクチャの研究の基盤を提供するためにマイクロカーネル自体に拡張性を持たせた．このマイクロカーネルをIntel486（100 MHz）上で実装した結果，throwのオーバヘッドは同一タスク内では0.89 μs，タスク間では113 μsとなった．

Development of an Extensible Microkernel that Utilizing a Single 2D Addressing Space

Tomoyuki Morinaga,† Eiichi Hayakawa,† Mitarou Namiki† and Nobumasa Takahashi†

A microkernel architecture is useful for pattern data processing since a proper resource management policy can be provided for each resource. However, conventional microkernels have problems such as the difficulty of linked data handling. Also, it is difficult to decide the trade-off between reliability and performance because the user can not define the policy of protection. We address the problems of the difficulty of linked data handling and protection of single addressing space with a single 2D addressing space. The single 2D addressing space consists of segments that have their own linear addressing space and are shared amongs all tasks. A construction primitive called throw is provided by this microkernel and dynamic linking is utilized for the extension of resource management, allowing high-performance and flexible system construction. This microkernel provides a protection space that enables to define the policy of protection by users and enables to decide the trade-off between reliability and performance. Moreover, to provide an experimental platform for a microkernel architecture, we give extensibility to the microkernel. This microkernel is implemented on an Intel 486 (100 MHz) processor. The throw within a task takes 0.89 μs and throws between tasks take 113 μs.

1. はじめに

計算機の扱う対象が多様化するにしたがって，OSで管理する資源が多様化し，対象に応じた効率の良い資源管理を行う必要がでてきている．

筆者らは，パターンデータ処理を指向したOSの開発を進めている．パターンデータの検索，認識，編集といった処理では，OSは時系列情報を含んだり，圧縮されたような様々なデータを扱う必要がある．多様なフォーマットのデータを効率良く処理するためには，資源管理の柔軟性が要求される．また，たとえば図形と手書き文字との合成といった処理によって生じるデータ間のリンクを容易に扱える基盤が必要である．

† 東京農工大学工学部
Faculty of Engineering, Tokyo University of Agriculture and Technology

マイクロカーネルアーキテクチャは資源管理を拡張でき，資源に適した管理ポリシーを提供できるので有効である．しかし，多重アドレス空間を提供するマイクロカーネルではタスクごとにアドレス空間が独立しているので，互いにリンクされたデータを扱うことは容易ではない．また，単一アドレス空間を提供するマイクロカーネルでは，データ間のリンクをポインタで表現できるものの，資源が線形なアドレス空間上にマップされるので，資源の大きさを超えたアクセスといった不正なアクセスを防ぐことが難しい．

マイクロカーネルアーキテクチャでは資源管理が拡張されるので，不正な拡張部からシステムを保護する必要がある．しかし，保護のオーバヘッドがあるので，信頼性と性能とのトレードオフが存在する．既存のマイクロカーネルでは保護のポリシーがマイクロカーネルで規定されていたので，ユーザがこのトレードオフを決定することはできなかった．信頼性を上げるためにスレッドごとに保護を行ったり，性能を上げるために複数のプロセスをまとめて保護するといった，信頼性と性能とのトレードオフをユーザが決定できる保護機構が必要である．

また，マイクロカーネルアーキテクチャの研究という立場からは，マイクロカーネル自体の拡張性が要求される．様々なOSの機能をどの階層でサポートすべきか，比較，実験を行える基盤が必要である．

これらの要求に応えるために，我々はリンクを持つデータの資源管理を可能にし，信頼性と性能とのトレードオフを図ることができる柔軟な保護機構を持つマイクロカーネルを開発した．このマイクロカーネルは拡張可能であり，構成プリミティブのオーバヘッドを低減することで拡張性の高い資源管理部の構成を可能にしている．

本論文では，次章で本マイクロカーネルのターゲットシステムであるOS/omicron第4版について述べ，3章ではマイクロカーネルの目的について述べる．4章ではマイクロカーネルの設計方針，5章では設計について述べ，6章では486プロセッサ上での実現について述べる．7章では関連研究について述べる．

2. ターゲットシステム

筆者らは，OS/omicron第4版[1]（V4）と呼ぶOSを開発している．V4の目的はイメージデータベースやイメージデータの認識，編集などのパターンデータ処理を統合する基盤を提供することである．

パターンデータの特徴は，多様なデータフォーマットが存在すること（多様性），データに対する解釈が場合によって異なること（多義性），データが互いに複雑なリンクを持つこと（関連性）である．

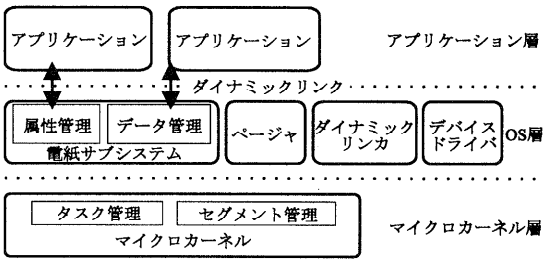


図1 OS/omicron第4版の全体構成
Fig. 1 Overview of the OS/omicron version 4.

パターンデータの一例として，我々は手書きインタフェースに着目している．手書きインタフェースでは，ストロークやビットマップ，ジェスチャなど多様なデータ型があり，ストロークは時系列情報を含んでいる．入力されたジェスチャの解釈は場合によって異なり，紙は貼り付けたり綴じたりすることによって互いにリンクを持つ．

V4では，入力された手書きデータに対して文字認識，図形認識など様々な処理が行われていく．また，認識などのデータ型変換や，手書き文字の図への貼付けといった編集処理ではデータ間のリンクが生成される．

パターンデータの型は多様であり，これをあらかじめシステムで規定することはできない．新しいデータ型に対してシステムが柔軟に対応するためには，ファイルシステムや描画系など，資源管理の様々な部分に対して拡張性が要求される．そこで，V4では電紙と呼ぶデータモデルを提供する．電紙は属性と呼ぶデータ処理手続きとデータを管理し，新しいデータ型とその処理手続きをユーザが拡張していく．

V4ではデータ間のリンクを管理するので，資源管理部でこのような処理を容易に記述できるようなサポートが必要である．また，新しいデータ型に対する資源管理を容易に追加できるようにするため，システムの拡張が必要となる．このようなことから筆者らはV4用マイクロカーネルを作成した．

図1に示すように，V4システムはアプリケーション層，OS層，マイクロカーネル層と呼ぶ3つの層から構成される．V4では，データの多義性の表現と資源管理の拡張にダイナミックリンクを，データ間のリンクをポインタで表現するためにワンレベルストアを採用する．

3. 本マイクロカーネルの目的

本マイクロカーネルの目的を次に示す．

(1) リンクを持つデータの処理に適したアドレス空間の提供

V4ではデータ間のリンクを管理する必要がある．多重アドレス空間を提供するOSではUNIX[2]のmmapに代表されるメモリマップの機構があるが，タスクごとにアドレス空間が独立しているのでポインタでデータ間のリンクを表現することは難しい．また単一アドレス空間モデルではメモリ上にマップされた資源の保護の問題がある．本マイクロカーネルでは，リンクを持つデータ処理に適したアドレス空間を提供する．

(2) 柔軟な保護機構の提供

既存のOSやマイクロカーネルでは保護のポリシーが固定されていたので，信頼性を上げるために細かい保護を行う，保護を行わずに性能を向上させるといった，信頼性と性能とのトレードオフを図ることが難しかった．そこで本マイクロカーネルでは，このトレードオフをシステムプログラマが図れる柔軟な保護機構を提供する．

(3) オーバヘッドの低減

マイクロカーネルの問題点の1つとして，オーバヘッドがあげられる．拡張性を上げるためにはシステムを細かくタスクに分割する必要があり，通信オーバヘッドが増加するという問題があった．V4では，データ型に応じた資源管理の拡張が必要であり，大容量のデータに対して認識や編集といった処理が行われるので，この問題は重要である．したがって，高速で拡張性の高いシステムの構成を可能にすることを目指す．

(4) マイクロカーネルの実験基盤の提供

システムに対する機能の追加方式には様々な方式があるが[3]，どれを選択するかはその機能のアプリケーションやマイクロカーネルとの通信頻度，保護の要求，必要とするデータ構造などを考慮して決定する必要がある．多様な資源管理について，複数の実装方式を比較，実験し，マイクロカーネルの持つべき機能を検証していくことは重要であると考える．したがって，本マイクロカーネルはマイクロカーネルアーキテクチャの実験基盤を提供する．

4. 設計方針

本マイクロカーネルでは，アドレス空間モデル，構成モデル，保護モデル，マイクロカーネル自体の拡張性の4つについて，次に示す設計方針をとった．

4.1 アドレス空間モデル

本マイクロカーネルは単一2次元アドレス空間と呼ぶアドレス空間モデルを採用する．このモデルでは，アドレス空間はセグメントidとオフセットによってアクセスされるセグメントの集合で構成され，すべてのタスクはこのアドレス空間を共有する．プログラミング言語におけるポインタは64ビット（セグメントidとオフセットそれぞれ32ビット）で構成される．

単一2次元アドレス空間モデルの特徴を次に示す．

(1) 資源をセグメントで表現しポインタで特定できる

単一なアドレス空間モデルなので，資源をセグメントで表現し，ポインタで特定することができる．また，データ間のリンクをポインタで表現することができる．

(2) 資源を不法なアクセスから保護できる

既存のアドレス空間モデルでは，資源が線形なアドレス空間にマップされるので，資源の大きさを超えたアクセスなどの不法なアクセスから資源を保護することが難しい．2次元アドレス空間モデルでは，セグメントは大きさやアクセスパーミッションといった保護情報を持つことができ，資源を不法なアクセスから保護することができる．

(3) メッセージコピーが必要ない

単一なアドレス空間なので，メッセージコピーの必要はなく，ポインタで通信できる．また，ポインタを含んだデータの処理も可能である．この特徴はパターンデータ処理を指向したV4のように，扱うデータの単位が大きいOSにとって有効であると考える．

4.2 構成モデル

マイクロカーネルが提供する構成モデルはシステムの柔軟性や性能を決定するので重要である．

従来のマイクロカーネルでは，メッセージで通信し合うタスクでシステムを構成する方法が一般的であった．しかし，このモデルでは拡張性を増大させるためには資源管理部を細かくタスクに分割する必要があり通信オーバヘッドが増大するという問題があった．

これに対して，本マイクロカーネルでは関数呼び出しによってマイクロカーネル上の資源管理部を構成し，ダイナミックリンクを用いて拡張する．拡張の単位とプロセッサの割当て単位であるタスクとを分離することによって，より高い拡張性と高速性を確保することができる．このモデルの特徴を次に示す．

(1) コンテキスト切替えの減少

アプリケーションからOSへ，OSからマイクロカーネルへシステムコールが発行される場合にも，コンテキストの切替えは発生せず，OSやマイクロカーネルはユーザのコンテキストで実行されるので，コンテキスト切替えの回数を減少させることができる．

(2) 拡張性の増大

プログラム中の識別子と実体とのバインディングは実

行時に決定されるので，関数のレベルでシステムを拡張することができ，従来のマイクロカーネルよりも高い拡張性を確保することができる．たとえば，ファイルシステムで新しいデータ型をサポートする場合には，ファイルシステム全体を交換することなく，データ圧縮機構などの必要な機能を追加するだけで対応できる．

ユーザのコンテキストでOSやマイクロカーネルが実行されるので，マイクロカーネルでは階層間の保護や例外処理のためのプリミティブを提供する方針をとる．また，構成プリミティブのインタフェースは，send/receiveといった特別なインタフェースを設けない．これによって，記述言語からパケットへの変換を行う際に発生するインタフェースのバグを減少させることができる．

4.3 保護モデル

単一アドレス空間モデルでは，アドレス空間が全システムで共有されるので保護が重要な問題となる．

既存のOSでは保護の対象が資源割当ての単位であるタスクやプロセスに固定されていたので，信頼性と性能とのトレードオフを図ることが難しいという問題があった．たとえば，デバッグ段階でタスクのある特定の手続きだけを保護したり，複数のタスクをまとめて保護することで保護のオーバヘッドを軽減するといったことができなかった．

そこで，本マイクロカーネルではプロセッサ割当ての単位であるタスクと保護の対象とを分離することで，システムプログラマが保護の対象を選択できるようにする．これによって，高速で動作する必要があるタスクをまとめて保護することで保護のオーバヘッドを減少させたり，信頼性を上げるためにある手続きだけを保護するといった，信頼性と性能とのトレードオフを図ることが可能になる．

4.4 マイクロカーネル自体の拡張性

マイクロカーネルアーキテクチャの実験基盤としてこのマイクロカーネルを使用するためには，マイクロカーネルの持つ機能自体を変更する必要がある．そこで筆者らはマイクロカーネル自体に拡張性を持たせることにした．拡張可能なマイクロカーネルを用いることで，専用プロセッサに対するコンテキスト管理の研究[4]やリアルタイム処理の研究において，どの層でどのような機能を提供すればよいのかといった検討を行うことができる．

マイクロカーネル自体が拡張可能であることから，マイクロカーネルに不正な拡張が行われた場合の保護の問題が発生する．しかし，マイクロカーネル自体を拡張可能にする目的はマイクロカーネルアーキテクチャの実験基盤を提供することである．我々は，マイクロカーネルはOSやマイクロカーネルの設計者によって拡張されるべきであるという立場をとり，マイクロカーネル自体の拡張に対する保護機構の導入は行わないという方針をとる．拡張方法には実行時の拡張，静的な拡張などがあるが，実験基盤として考えた場合，実行時の拡張を行う必要はない．また，静的拡張ではコンパイル，リンク，インストールなどの手間が増加するので，本マイクロカーネルではブートストラップ時にマイクロカーネルをリンクする方式を採用する．

5. 設　　計

ここではマイクロカーネルの設計について述べる．

5.1 マイクロカーネルの全体構成

図2にマイクロカーネルの全体構成を示す．

マイクロカーネル上のV4の資源管理部は，図2のインタフェース部が提供するプリミティブを用いて記述され，このプリミティブが変更されることはない．しかし，本マイクロカーネルをマイクロカーネルアーキテクチャの実験基盤として用いる場合，マイクロカーネル自体の拡張性が必要になるので，マイクロカーネルをインタフェース部と仮想マシン部と呼ぶ2つの層に分割する設計とした．

マイクロカーネル自体の拡張性を確保するために，マイクロカーネル自体もポリシーとメカニズムを分離する構成とし，仮想マシン部ではメカニズムを，インタフェース部ではポリシーを実現する．これらの2つの層はブートストラップ時にリンクされるので，インタフェース部を取り替えることでマイクロカーネルを拡張することができる．

それぞれの層の設計を次に示す．

- 仮想マシン部

コンテキスト管理，ハードウェア割込み，セグメント管理のプリミティブを提供する．これらのプリミティブはポリシーを含んでいない．たとえば，仮想マシン部ではコンテキスト切替えのプリミティブは提供されるが，スケジューリングポリシーは定義されていない．仮想マシン部で低レベルなプロセッサに依存した機能

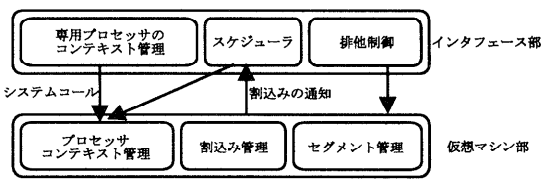


図2 マイクロカーネルの全体構成
Fig. 2 Overview of the microkernel.

を仮想化し，インタフェース部でのより高度な抽象化を可能にする．

• インタフェース部

仮想マシン部が提供するプリミティブを用いて，資源管理のポリシーやモデル，インタフェースが実現される．スケジューリングや排他制御といったタスク管理，構成モデル，保護機構などがこの層で実現される．V4の資源管理はこのインタフェース部の提供するプリミティブを用いて記述されるが，他のマイクロカーネルのモデルを実現する場合にはこのインタフェース部を変更することによって対処できる．

この2つの層は関数呼び出しを用いて通信するので，オーバヘッドを増加させることなくマイクロカーネル自体の拡張性を確保することができる．

5.2 セグメント管理

本マイクロカーネルでは，単一2次元アドレス空間モデルを採用する．このモデルでは，資源はすべて2次元アドレス空間へマップされ，セグメントとして提供される．これによって，すべての資源はポインタで特定することができ，すべての資源をセグメントアクセスの保護機構によって保護できるという利点がある．

セグメントは次に示す情報を持つ．

(1) 属性

マイクロカーネルは様々な資源をセグメントとして提供するので，資源を区別する情報として属性を定義する．資源の型は属性によって指定される．

(2) アクセス権

アクセス権はセグメントに対するアクセスパーミッションを定めるものである．アクセス権は読み出し可能，書込み可能，実行可能の組合せである．アクセス権は後述する保護空間によって定義される．

(3) 大きさ

それぞれのセグメントには大きさが定義されており，この大きさを超えたアクセスは不正なアクセスとなる．

(4) システムレベル

システムレベルは，セグメントがどの層（マイクロカーネル，OS，アプリケーション）に属するかを決定するもので，階層間の保護を行うために提供する．

5.3 構成モデル～throw

本マイクロカーネルは，マイクロカーネル上の資源管理部を拡張するために，throwと呼ぶ拡張性と性能とのトレードオフを低減する構成モデルを提供する．このモデルでは，OSとアプリケーションはモジュールから構成される．このモジュールは1つ以上の手続きの集合からなり，プロセッサ割当ての単位であるタスクとは独立している．本マイクロカーネルはプロセッサ割当て単位としてタスクを提供する．モジュールがタスクから分離されているので，同一タスク内でモジュール間の通信が行われる場合にはコンテキスト切替えは発生しない．また，システムコールを発行した場合にもコンテキストの切替えは発生せず，呼び出し側のコンテキストでOSが実行される．したがって，コンテキスト切替えの回数を減少させることができる．

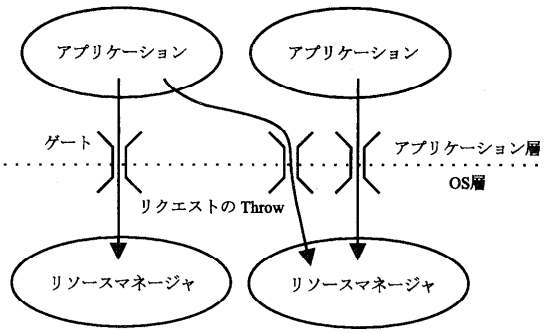


図3 システムレベル間の throw
Fig. 3 Inter-systemlevel throw.

しかし，コンテキストが切り替わらないので，階層間の保護や，タスク間通信，例外処理などが必要となる．そこで，本マイクロカーネルはこれらのプリミティブを提供する．

throwのインタフェースはシステム記述言語（言語C）と同じである．タスク間通信のインタフェースも関数呼び出しのインタフェースとなっている．これによって，プログラミング言語からメッセージパケットへと変換を行う際に発生するインタフェースのバグを低減できると考える．

マイクロカーネルは次のようなthrowを提供する．

(1) モジュール間

すべてのCのプログラムは関数からなり，処理は他の関数へとリクエストをthrowすることで行われる．リクエストをthrowすることは関数を呼び出すことと同じであり，throw先はコードセグメントである．

(2) システムレベル間

リクエストがシステムレベルをまたぐ場合（ユーザからOSへ，OSからマイクロカーネルへ）には，ゲートに対してthrowする．ゲートはシステムコールのエントリを定義しており，より高い特権の手続きやデータの不正なアクセスを防止する（図3）．

(3) タスク間

ほかのタスクへとリクエストをthrowすることも可能である．この場合，プログラムはマイクロカーネルに対してエントリポイントと処理タスクのTCB（Task Control Block）へのポインタを組にして登録してお

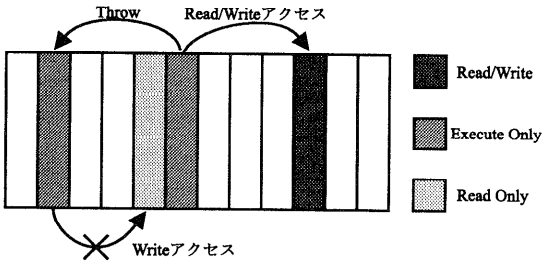


図4 保護空間
Fig. 4 Protection space.

く．このゲートに対してリクエストをthrowすることによって，リクエストをほかのタスクによって処理させることができる．

(4) 割込みとフォールト

ハードウェア割込みやフォールトはthrowによってOSに通知される．リクエストはマイクロカーネルによってゲートまたはエントリポイントに対してthrowされ，割込みハンドラやフォールトハンドラがマイクロカーネルによってディスパッチされる．

(5) 例外処理

本マイクロカーネルでは，ユーザタスクのコンテキストでOSが実行されるので，タスクを消去したりその終了処理を行うことが難しい．終了処理はタスクの実行軌跡をたどって行う必要がある．そこで本マイクロカーネルでは例外処理のためのthrowを提供する．このthrowはUNIXのsetjmp/longjmpと似ているが，保護空間をサポートしているという点が異なる．

throw先はプログラムの実行時に，動的にダイナミックリンカによって決定され，必要に応じてセグメントやゲートが作成される．

5.4 保護空間

単一アドレス空間モデルでは，メモリアクセスの保護が重要となる．本システムは3つの層から構成されるが，それぞれの階層はシステムレベルによって保護される．さらに，マイクロカーネルでは，同一システム階層内での保護機構として，保護空間と呼ぶセグメントのアクセス保護機構を提供する．ここでは，この保護空間の設計について述べる．

保護空間は，アクセス可能なセグメントが定義された仮想的な2次元アドレス空間であり，アクセス権は書込み可能，読み出し可能，実行可能のそれぞれの組合せである．ある保護空間からは，その保護空間でアクセスが許可されたセグメントにしかアクセスすることができない（図4）．

保護空間の特徴は，プロセッサ割当ての単位であるタスクとは独立した保護機構であり，保護のポリシーをシステムプログラマが設定できる点である．従来のシステムでは保護の対象が固定されていたので，信頼性と性能とのトレードオフが存在した．保護空間では，保護のポリシーをユーザが設定できるので，このトレードオフを軽減することができる．

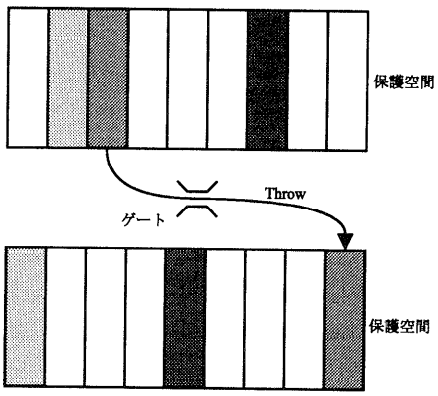


図5 保護空間の間でのthrow
Fig. 5 Inter-protection space throw.

タスクは，マイクロカーネルが提供するゲートを用いて，ほかの保護空間へと移動することができるので，1つのタスク内でのコードやデータの保護が可能になる．また，複数のタスクをまとめて保護することも可能である．ほかの保護空間へ移動するにはマイクロカーネルが提供するゲートに対してリクエストをthrowする（図5）．

保護空間の割当てポリシーはシステムプログラマが定義できるので，信頼性と性能とのトレードオフを柔軟に決定することができる．たとえば，次のような保護空間の割当てポリシーを定義することができる．

(1) タスク間

それぞれのタスクに対して保護空間を割り当てれば，タスク間の保護が可能になる．これは多重アドレス空間モデルと同じ効果をもたらす．

(2) モジュール間

保護空間をそれぞれのモジュールに割り当てることで，モジュール間の保護を行うことができる．

(3) スレッド間

それぞれのスレッドに対して保護空間を割り当てれば，スレッドごとの保護を行うことができる．

throwのインタフェースは関数呼び出しと同じなので，保護空間の割当てをプログラムを書き直すことなく変更できる．モジュールをデバッグしているときには，そのモジュールに対して保護空間を割り当てることで，ほかのモジュールに対する不正なアクセスを防止するといったことが可能である．

6. 実現と評価

本マイクロカーネルをインテルの486プロセッサ上で実現した．マイクロカーネルはプロセッサのプロテクトモードを利用しており，システムレベルはリングプロテクションを，システムレベル間のthrowはコールゲートを用いて実現した．システムにおけるすべての通信はthrowによって実装しており，throwはコールゲートまたはcall命令を用いて実装しているので，本実装では割込みやトラップ命令は用いていない．

このマイクロカーネルは言語Cを用いて記述しており，このCコンパイラ[5]は我々の研究グループで開発された2次元アドレスとダイナミックリンクに対応したものである．

表1にハードウェアの構成を，**表2**にマイクロカーネルの実現規模を示す．マイクロカーネルのサイズは約6000行で175kバイトとなった．

仮想マシン部とインタフェース部は同じシステムレベルに配置されているので，この2つの層の間ではシステムレベルの変更が起こらず，マイクロカーネルを2つの層に分けたことによるオーバヘッドは発生していない．

6.1 仮想マシン部の実現

仮想マシン部の実現規模は3500行であり，ディスパッチャ以外はほとんど言語Cで記述してある．

仮想マシン部のシステムコールの速度を**表3**に示す．コンテキストスイッチにかかる時間は10.7 μsであり，インタフェース部に対する割込みの通知は，割込みの発生からエントリ関数へと制御が移るまでで2.4 μsであった．

6.2 インタフェース部の実現

インタフェース部の実現規模は，言語Cで約3000行となった．

本マイクロカーネルと他のマイクロカーネル[6]のシステムコールの時間を**表4**に示す．throwのオーバヘッドは呼び出しを行ってから呼び出し元へと制御が帰ってくるまでであり，呼び出された側では即座にリターンした場合を測定した．また，他のマイクロカーネルのシステムコールの時間は，0バイトのメッセージ通信を行った場合である．なお，set_throw/do_throwは次のプログラムを用いて測定した．

```
if(set_throw() == 0) {
  do_throw(1);
} else {
  ;
}
```

この結果，set_throwとdo_throwに4.6 μsかかった．1回目のset_throwの呼び出しは2 μsであった．通常のプログラムの実行ではset_throwは1回しか呼び出されないので，実用的な速度であると考える．

システムレベル間のthrowは，従来のシステムコールのオーバヘッドに対応するものであり，そのオーバヘッドは1.2 μsであった．

本マイクロカーネルでは，プロセッサ割当ての単位であるタスクとモジュールとは分離されているので，モジュール間通信ではコンテキスト切替えは起こらない．このモジュール間throwのオーバヘッドは0.89 μsであった．これはMach[7]のRPCと比べると100倍，L3[8]と比較しても10倍高速である．これによって，柔軟で効率の良いオペレーティングシステムを作成することができると考える．しかし，タスク間のthrowは113 μsかかっている．これは，次の要因によると

表1 マシンのスペック
Table 1 Machine specification.

プロセッサ	Intel 486DX4（100 MHz）
メモリ	16 M bytes

表2 マイクロカーネルのステップ数
Table 2 Code steps of the microkernel.

処理内容	仮想マシン部		インタフェース部	
	言語C	アセンブラ	言語C	アセンブラ
初期化	337	96	56	0
タスク管理	337	244	1265	0
メモリ管理	211	0	543	0
割込み管理	256	794	374	0
その他	1296	0	830	0
合計	2437	1134	3068	0

表3 システムコールのオーバヘッド（仮想マシン部）
Table 3 Systemcall overhead (virtual machine layer).

処理内容	時間（μs）
コンテキスト切替え	10.7
割込み通知	2.4

表4 システムコールのオーバヘッド（インタフェース部）
Table 4 Systemcall overhead (interface layer).

処理内容	時間（μs）
throw（モジュール間）	0.898
throw（システムレベル間）	1.28
throw（タスク間）	113
非同期throw（タスク間）	124
throw（タスク間，割込み）	102
throw（割込み）	12.4
set_throw/do_throw	4.60
Mach（486 50 MHz）	230
L3（486 50 MHz）	10
SPIN（Alpha 21064 133 MHz）	102

考える．

(1) コンパイラの最適化が十分でない

我々が現在用いている言語Cコンパイラは十分な最適化を行っていない．特にセグメントレジスタへのアクセスをともなうマルチセグメント命令の最適化は有効であり，手作業の最適化でタスク間throwの速度を，セグメントプレフィックスバイトの削除によって20 μs，不要なセグメントレジスタのpush/popの削除によって40 μs高速化できた（DX2 40 MHz）．

(2) セグメント操作命令が遅い

インテル系のプロセッサでは，マルチセグメント命令は極端に遅い．486では汎用レジスタのpop命令が1クロックなのに対して，セグメントレジスタのpopは9クロックと9倍の時間がかかる．

7. 関連研究

ここではアドレス空間モデル，構成モデル，保護モデル，マイクロカーネル自体の拡張性といった点から関連研究について述べる．

7.1 アドレス空間モデル

代表的なマイクロカーネルであるMachやChorus[9]，Spring[10]は，多重アドレス空間モデルを採用している．このモデルでは，タスクごとにアドレス空間が独立しているので，リンクを持つデータを扱うことが難しい．また，通信オーバヘッドの問題もある．

Lucas[11]やDSR[12]では，多重アドレス空間モデルでデータ間のリンクをポインタで表現することを可能にしている．これらのシステムでは，ポインタを含んだデータを特定のアドレスへマップする機構を備えており，マップするアドレスがすでに使用されていた場合にリロケーションを行う．

Opal[13]，Cubix[14]などの単一アドレス空間モデルを採用しているマイクロカーネルでは，全タスクは1つのアドレス空間を共有するのでデータ間のリンクをポインタで表現できるものの，アドレス空間にマップされた資源の保護の問題がある．これをどのように解決するかが課題である．

Multics[15]は2次元アドレス空間を採用しているが，単一ではなくプロセスごとに2次元アドレス空間が割り当てられる．セグメントは，複数のプロセス間でそれぞれ違ったセグメントidで共有される．

Exokernel[16]やSPIN[17]では，アドレス空間モデルをカーネルではなくユーザが定義できる．これらのマイクロカーネルでは多重アドレス空間，単一アドレス空間のどちらのモデルも実装することができるが，どちらのモデルも上記の問題点を持っている．

これらに対して，本マイクロカーネルでは単一アドレス空間によってポインタでリンクされたデータ構造の表現を容易にするとともに，2次元アドレスによって保護の問題を解決している．

また，ObjectStore[18]，O_2[19]，EXODUS[20]といったオブジェクト指向データベースでは永続オブジェクトをアクセスできるデータベースアクセス言語をサポートし，ワンレベルストアが実現されている．特にObjectStoreでは，永続オブジェクトをC++言語から通常のオブジェクトと同様にアクセスできる．これに対して我々のシステムでは，マイクロカーネルのアドレス空間のレベルからワンレベルストアをサポートしており，このようなデータベースシステムはマイクロカーネル上のサーバとして実装できると考えている．

7.2 構成モデル

大部分のマイクロカーネルでは，メッセージ通信に基づいた構成モデルを採用している．しかし，このモデルの問題点としてあげられるのは，コンテキストスイッチと，多重アドレス空間モデルでのアドレス空間切替えによるオーバヘッドである．通常のマイクロカーネルではサーバ間の通信時に4回のコンテキストとアドレス空間の切替えが発生し，メッセージのコピーも最低1回は発生することになる．

もう1つの問題点としてあげられるのは，プログラミング言語とマイクロカーネルインタフェースの間のギャップが大きく，プログラミング言語の手続き呼び出しからパケットへと変換を行う必要があるという点である．この変換はオーバヘッドやインタフェースのバグの原因となる．このためMachでは，MIG[21]と呼ばれるインタフェースジェネレータを提供しているが，MIGとソースプログラムの2つを記述する必要がありバグの原因になるという問題がある．

本マイクロカーネルでは，関数呼び出しを用いることでコンテキスト切替えのオーバヘッドを減らし，システム記述言語と等価なインタフェースにすることでインタフェースのバグの発生を減少させている．

7.3 保護モデル

多重アドレス空間モデルを採用するMachやMulticsなどでは，アドレス空間をタスクやプロセスごとに割り当てることで，保護を実現している．

また，単一アドレス空間モデルを採用しているOSでは，アドレス空間が全システムで共有されるので，プロテクションドメインと呼ばれる保護の機構を提供している．しかし，Opalではプロテクションドメインは UNIX のプロセスからアドレス空間を分離した概念なので，保護はプロセス単位に固定されている．

Chorus ではカーネルアドレス空間内で動作する supervisor actor をサポートしているが，これはカーネルに特化した機能であり，本マイクロカーネルの保護空間とは異なる．Cubix では，プロテクションドメインは ACL（Access Control List）によって実装され，スレッドはプロテクションドメイン間を移動することができる．筆者らのマイクロカーネルと Cubix の相違点は，スレッドが保護空間を移動するためのインタフェースを関数呼び出しに統合している点，保護空間の実現をプロセッサの MMU（486 の LDT）で行う点である．

筆者らは，保護空間というシステムプログラマが保護のポリシーを決定できる保護機構を提供することによって，タスク間，手続き間といった多様な保護を行えるようにし，信頼性と性能とのトレードオフを図ることを可能にしている．

7.4 マイクロカーネル自体の拡張性

Mach，Opal など大部分のマイクロカーネルでは，マイクロカーネル自体は拡張可能にはなっていない．しかし，マイクロカーネルで固定した機能を提供するアーキテクチャには限界がある．

Exokernel では，カーネル自体はハードウェアの保護と多重化だけを行い，それ以外の機能はアプリケーションにリンクされるライブラリとして実現される．また，SPIN はカーネル自体が拡張可能である．カーネルの記述には Modula-3 言語を用い，言語レベルでのプロテクションとランタイムのチェックを行っている点が特徴である．Exokernel，SPIN 自体はアドレス空間モデルや構成モデルを規定しておらず，上述した各モデルに対する解決方法を提案していないという点が課題であると考える．

8. おわりに

本論文では，単一2次元アドレス空間モデルを採用した拡張可能なマイクロカーネルについて述べた．

このマイクロカーネルでは，単一2次元アドレス空間モデルを採用することで，互いにリンクされたデータの扱いを可能にし，単一アドレス空間で問題となる保護の問題に対応した．また，throw と呼ぶ拡張性と性能とのトレードオフを軽減させる構成プリミティブを提案した．本マイクロカーネルのシステムコールのオーバヘッドは 1.28 μs であり，既存のマイクロカーネルよりも大幅に高速化することができた．また，ユーザが信頼性と速度のトレードオフを柔軟に図ることができる保護機構である保護空間の設計を行った．

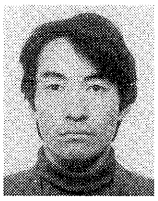
現在，このマイクロカーネル上でウィンドウシステム "未"[22] とコマンドインタプリタが動作している．今後は，仮想記憶と保護空間を実現して評価を行う．また，V4 のファイルシステムやページャ，ダイナミックリンカなどの資源管理部についても実現を行っていく予定である．

参考文献

1) Hayakawa, E., Namiki, M. and Takahashi, N.: *Basic Design of SHOSHI Operating System that Supports Handwriting Interfaces*, 情報処理学会論文誌, Vol.35, No.12, pp.2590–2601 (1994).
2) 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 (1989).
3) 徳田英幸：分散リアルタイムシステムのための OS アーキテクチャ，情報処理，Vol.35, No.1, pp.18–25 (1994).
4) 中原雅彦，早川栄一，岡野裕之，並木美太郎，高橋延匡：複数の浮動小数点表現法を処理するシステム環境の設計と実現，情報処理学会論文誌，Vol.33, No.4, pp.481–190 (1992).
5) 並木美太郎，中村浩之，田中広幸，森永智之，加藤泰志，早川栄一，高橋延匡：2次元アドレスとダイナミックリンクのための実行コンテキストの設計と言語 C 処理系の開発，情報処理学会研究会報告，95-OS-69, pp.31–36 (1995).
6) Liedtke, J.: On μ-Kernel Construction, *Proc. 15th ACM Symposium on Operating System Principles*, pp.237–250 (1995).
7) Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: Mach: A New Kernel Foundation For UNIX Development, *USENIX Summer '86*, pp.93–112 (1986).
8) Liedtke, J.: Improving IPC by Kernel Design, *Proc. 14th ACM Symposium on Operating System Principles*, pp.175–188 (1993).
9) Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P. and Neuhauser, W.: Overview of The Chorus Distributed Operating System, *Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp.39–69 (1992).
10) Hamilton, G. and Kougiouris, P.: The Spring Nucleus: A Microkernel for Objects, *Proc. 1993 Summer USENIX Conference*, pp.147–160 (1993).
11) 猪原茂和，上原敬太郎，宮澤　元，益田隆司：オペレーティングシステム Lucas における 64 ビットアドレス空間の管理，情報処理学会研究会報告，

93-OS-61, pp.81–88 (1993).
12) 松原克弥，加藤和彦：分散仮想記憶技術を用いた分散共有格納庫システムの実現法について，情報処理学会研究会報告，94-OS-65, pp.153–160 (1994).
13) Chase, J., Levy, H., Baker-Harvey, M. and Lazowska, E.: Opal: A Single Address Space System for 64-bit Architectures, *The Third Workshop on Workstation Operating Systems*, pp.80–85, IEEE Computer Society Press (1992).
14) 岡本利夫：単一仮想記憶空間を特徴とするオペレーティングシステムについて，コンピュータシステム・シンポジウム，pp.39–46 (1995).
15) Corbató, F.J. and Vyssotsky, V.A.: Introduction and Overview of the Multics System, *Proc. AFIPS, FJCC,* Vol.27, pp.185–196 (1965).
16) Engler, D.R., Kaashoek, M.F. and O' Toole Jr., J.: Exokernel: An Operating System Architecture for Application-level Resource Management, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.251–266 (1995).
17) Bershad, B.N., et al.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.267–284 (1995).
18) Lamb, C., Landis, G., Orenstein, J. and Weinreb, D.: The ObjectStore Database System, *Comm. ACM*, Vol.34, No.10, pp.50–63 (1991).
19) Deux, O., et al.: The O_2 System, *Comm. ACM*, Vol.34, No.10, pp.34–48 (1991).
20) Hanson, E.N., Harvey, T.M. and Roth, M.A.: Experiences in DBMS Implementation Using an Object-oriented Persistent Programming Language and a Database Toolkit, *OOPSLA*, pp.314–328 (1991).
21) Boykin, J., Kirschen, D., Langerman, A. and LoVerso, S.: *Programming under Mach*, Addison-Wesley (1993).
22) 早川栄一，河又恒久，宮島 靖，加藤直樹，並木美太郎，高橋延匡：ペンインタフェース研究・開発のためのウィンドウシステム“未”（HITSUJI）の設計と実現，情報処理学会論文誌，Vol.36, No.4, pp.932–943 (1995).

(平成8年7月10日受付)
(平成9年3月 7 日採録)



森永 智之（学生会員）

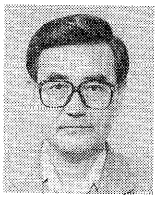
1969年生．1993年東京農工大学工学部電子情報工学科卒業．1995年同大学大学院電子情報工学専攻博士前期課程修了．同年同大学大学院電子情報工学専攻博士後期課程進学，現在に至る．オペレーティングシステムに興味を持つ．

早川 栄一（正会員）

1989年東京農工大学工学部数理情報工学科卒業．1992年同大学大学院電子情報工学専攻博士後期課程単位取得退学．同年同大学電子情報工学科助手．博士（工学）．オペレーティングシステムおよび日本語情報処理の研究に従事．ACM，IEEE，電子情報通信学会各会員．

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．博士（工学）．オペレーティングシステム，プログラミング言語，ウィンドウシステム，並列処理，コンピュータネットワークおよび日本語情報処理の研究・開発に従事．ACM，IEEE，電子情報通信学会各会員．



高橋 延匡（正会員）

1957年早稲田大学第一理工学部数学科卒業．同年4月（株）日立製作所中央研究所入社，HITAC5020モニタ，TSSの開発などに従事．1977年東京農工大学工学部数理情報工学科教授．1989年電子情報工学科教授．理学博士．オペレーティングシステム，日本語情報処理などの研究，教育に従事．