オペレーティング・システム 55－4
（１９９２．６．８）

単一仮想記憶域を提供する64ビット・アドレス指向OS

瀬川英生 野末浩志 申承昊 岡本利夫
前田賢一 斉藤光男
(株) 東芝 総合研究所

分散処理システムの発達はクライアント／サーバ型プログラミング・スタイルの高まりをもたらした。クライアント／サーバ型プログラミングはマルチプロセッサ型アーキテクチャにはふさわしいものの、カーネルを介したプロセス間通信のコストが高いという欠点を有している。

現在設計中の64ビット・アドレス指向OS CUBIXではこの高い通信コストを単一仮想記憶域という新しいメモリ・モデルの導入により解決している。CUBIXではクライアント・プログラムとサーバ・プログラムをひとつの仮想空間にロードし、空間切替をせずにサブルーチンコールのようにプロセス間通信を実行する。また、このような機能を実現するため、多重仮想空間によるメモリ保護の代わりに単一仮想空間内の細粒度のメモリ保護機能を提供するアクセス制御リストに基づくメモリ管理装置を開発する。

64bit Address-Oriented Operating System Supporting SVS

Hideo Segawa, Sung Ho Shin, Hiroshi Nozue, Toshio Okamoto
Ken-ichi Maeda, Mitsuo Saito
Toshiba Research and Development Center.

The development of distributed sytems has brought popularization of a client-server programing style, which well fits multiprocessor architecture but limits the performance by the cost of invoking kernel-based interprocess communication.

The newly designed operating system CUBIX solves this difficulty with a new memory model of single virtual storage. CUBIX loads both client and server programs in a single virtual space(SVS) and executes interprocess communication like a subroutine call without switching the SVS. Besides, an ACL-based memory management unit is also proposed to realize fine-grained memory protection within a SVS instead of multiple space switching memory protection.

1 はじめに

現在、標準OSとして流通しているUnix[1]の最初のバージョンは1969年、ベル研で開発された(Leffler et al.[1])。当初はPDP-11上で開発されたため、16ビットの小さなアドレス空間しか持たなかったが、その後、1979年にVAX-11/780に移植され、32ビットの仮想空間がもたらされた。

Unixは20年以上も前になされた基本設計のうえに、様々な機能が加えられ発達してきた。そして、近い将来、MIPSのR4000をはじめとして64ビット・プロセッサが市場に登場する(Mashey[2])。Unix本来のプロセスや仮想空間のモデルをそのまま64ビットアドレス空間に拡張することは

- 特殊な科学技術計算やCADプログラムを除いて、通常のプログラムは32ビットのアドレス空間があれば実行可能である。

- ネットワーク環境が発達し、クライアント・サーバによるプログラミング・スタイルが一般化しつつある。クライアント・プログラムとサーバ・プログラムの通信はパイプやソケット、RPC(リモート・プロシージャ・コール)といったUnixのプロセス間通信機能を利用するが、この機能はカーネルを介するためオーバヘッドが大きい。そこで、複数のプロセスからアクセス可能な共有メモリ機構が導入され、それを利用したユーザ・レベルのプロセス間通信などが研究されている(Bershad et al.[3])。

など、必ずしも適当ではない。

本報告では64ビットアドレス空間に対応した新しいメモリ・モデルを有するオペレーティング・システムCubix[2]の構想をまとめた。Cubixは単一仮想記憶域(SVS:Single Virtual Storage)を特徴としており、サーバ・プログラムとクライアント・プログラムを同一の仮想空間に格納し、空間切替えなしに通信する。

[1]UNIXはBell研が開発し、ATTがライセンスするOS

[2]CUBe of 2 byte unIX

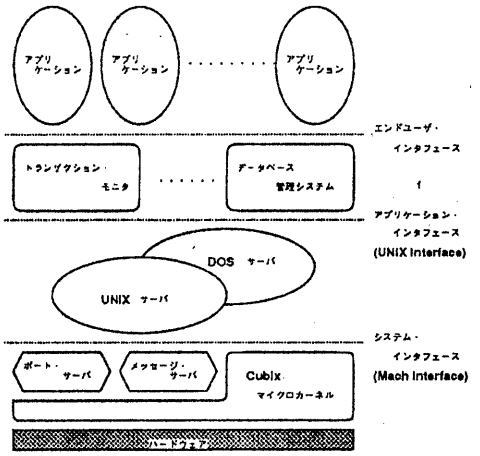

図1: Cubixマイクロカーネルの構成

また、近年、ネットワークや共有メモリ型マルチプロセッサの発達にともない、Machなど分散指向のOSが数多く登場している(Accetta et al.[4])。Cubixではそれらの成果も積極的に採り入れ、マイクロカーネルによるネットワーク透過な分散システムの実装を目指している(Okamoto et al.[5])。

2 Cubixの概要

2.1 マイクロカーネルで扱う資源

Cubixマイクロカーネルでは

- スレッド プログラムの実行主体であり、プログラム・カウンタ、汎用レジスタなどが含まれる。

- メモリ・セクション SVSにおけるひとまとまりのメモリ領域でその実体(バックアップ)であるディスクなどの記憶域を含む。物理メモリはキャッシュとして作用する。

と2種の資源を考えている。

マイクロカーネルでは上位層でこれらの資源を扱うためのシステムコールを提供する。さらにそれらを用い、ユーザ・レベルのサーバではユーザにとってより扱いやすい以下のようなインタフェースを提供する。

- プロシージャ Cubixにおけるプログラムの最小実行単位で、テキスト・セクション(T)、データ・セクション(Dセクション)とスタック・セクション(Sセクション)によって構成される。

- ファイル ファイル・サーバによって管理されるDセクション。

- プロシージャ間通信(IPC) IPCはプロシージャ間にまたがるユーザ・レベルのjumpおよびreturn機能で、IPCサーバによりIPC実行権が管理される。

Cubixの抽象化はMachのそれをほぼ踏襲している。CubixとMachの対照は以下の通り。

- CubixのスレッドとMachのスレッドはほぼ同一で、処理の制御の流れの単位。Machのスレッドは1タスク空間内で閉じているが、Cubixのスレッドは複数のプロシジャを実行できる。

- Cubixのメモリ・セクションはMachのメモリ・オブジェクトとほぼ同様で、一元化仮想記憶域(OLS:One Level Storage)である。ただし、Cubixは単一仮想空間であり、全てのセクションに割り付けられているアドレスは唯一であるが、Machは多重空間であり、空間ごとに独立してアドレスが割り付けられる。

- Machのメッセージ・パッシングのための資源であるポートとメッセージはCubixでは扱わない。ユーザレベルのIPC機能がCubixの特徴であり、ポート・サーバおよびメッセージ・サーバがそれらの機能を提供し、Machをエミュレートする。タスクに関しても同様である。

Cubixは図1のようにMachカーネル・インタフェースとUnixカーネル・インタフェースを守っており、デファクト・スタンダードなOSとの互換性が保てるよう配慮してある。

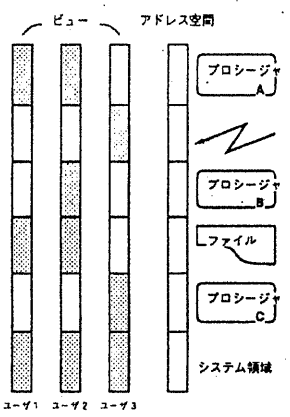

図 2: 単一仮想空間とビューの関係

2.2 プログラミング・モデル

Unixはシングルプロセッサを想定したOSであり、ひとつのプロセッサが独立したアドレス空間を有する一つのプロセスを実行するプログラミング・モデルに基づいている。Machはマルチプロセッサ向きにプロセスをタスクとスレッドに分解し、複数のスレッドが一つのがタスク内を並行実行できるように拡張した。Cubixのプログラミング・モデルでは複数のメモリ・セクション上を複数のスレッドが独立して並列処理する。あるスレッドからアクセス可能なセクションの集合、すなわち、

- あるTセクションからデータ・アクセス可能なD、SまたはFセクション。Unixにおけるプロセス、およびファイルのmmapに相当する。

- あるTセクションから分岐可能なTセクション。Unixにおける RPC に相当する。

をビューと呼ぶ。つまり、スレッドはビューの指定するメモリ領域をひとつのプログラムのように実行することになる。

Cubixのプログラミング・スタイルは

- IPCはプロシージャの切替を行なわず、サブルーチン・コールのようにジャンプする。このようなモデルは処理の意味的な流れを考えた時に自然で

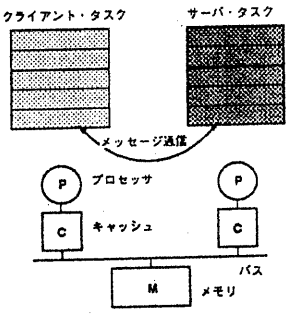

図 3: Mach のプログラム実行形式

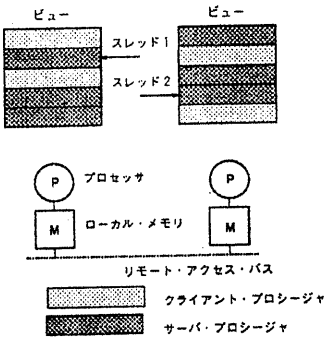

図 4: Cubix のプログラム実行形式

あり、また、クライアント・サーバ・プログラミングの高速化に役立つ。

- ファイルの mmap や共有メモリといった Unix 本来のモデルからははずれているが、後から高速化のために導入された概念もモデル化される。

など、より分散環境に適し、かつ、Unix の実情と整合したモデルと考える。

2.3 高速 IPC とキャッシュに関する考察

Mach におけるクライアント・サーバ・プログラムの実行形式を考える。Mach では処理が複数のプロセッサによって構成される並列・分散環境を想定しており、

- クライアントおよびサーバ・タスクをプロセッサに固定。
- タスク間通信機能によりメッセージをクライアント・タスクからサーバ・タスクに転送。

といった処理形式となる。ところが、単一のプロセッサでこのクライアント・サーバ型の処理を行なう場合、クライアント・タスクとサーバ・タスクがコンテクスト・スイッチを起こしながら処理を進めていくことになり、キャッシュの効果もなかなか得られにくい。複数のプロセッサが利用できる環境でもなるべくコンテクスト・スイッチをせずに、特定のプロセッサに特定のタスクを固定したまま、サービスに応じてメッセージの送信先タスクを選択するような実行形式がもっとも効率がよい。

一方、Cubix では

- スレッドをプロセッサに固定。
- 複数のプロシージャをひとつのアドレス空間に取り込み、なるべくプロセッサ間通信を使わず、できる限りひとつのプロセッサ上で関連した処理を行なう。

といった処理形式をとる。複数プロセッサ上でサーバ・クライアント処理を行なう場合のオーバヘッドは Mach と同一である。しかし、単一プロセッサ上ではコンテクスト・スイッチを起こさないためクライアント・プロシージャおよびサーバ・プロシージャ双方のプログラムのうちよく使われるラインのみがキャッシュに残り、Mach に較べ性能が向上する。

3 メモリ・モデル

Cubix では SVS により、すべてのプロシージャとファイルが同一の仮想空間に格納され、

- 空間切替えせずに IPC が jump 命令で実現可能。
- すべてのファイルがあらかじめマップされており、高速アクセス可能。

といった高速性に利点がある。

一方、単一仮想空間を採用したことで

- Unix や Mach がアドレス空間を切替えることで実現していた不正アドレスアクセス防止機能の実現。
- 多種多様なマイクロプロセッサへの移植性や Unix や Mach など既存標準 OS との互換性。

などが新たな課題となる。

3.1 単一仮想記憶域

Cubix の単一仮想空間は多重仮想空間と異なり、以下のような特徴を有する。

- すべてのプロシージャおよびファイルは唯一のアドレス空間に割り付けられるため、共有メモリやライブラリはあらゆるスレッドからも同一のアドレス位置に割り付けられる。
- プロシージャはロード時にフリー・メモリ・セクションの中から適当な仮想アドレス位置を選び、割り付ける。ファイルも create 時に仮想アドレス位置が決定される。

また、Cubix ではスレッド切替えする時に、仮想アドレスをリマップせずにビューを切替える。それは、

- 一般に物理キャッシュに較べ論理キャッシュは大容量が実装可能で、高速化をはかりやすい。単一仮想空間ではある仮想アドレスに対する実体は唯一であり、論理キャッシュの特性を生かしやすい。
- Unix において空間切替は高価な処理であるといわれているが、カーネル・コードの実行そのものよりも複数のプロセス間で共通のアドレスを有しているために生ずるキャッシュの整合性操作に大きな時間がかかっている (Mogul et al.[6])。単一仮想空間ならばアドレス空間は唯一であり、そのような問題は生じない。

などの理由からキャッシュの効果を得やすく、プログラムの高速実行にに有利と判断したためである。

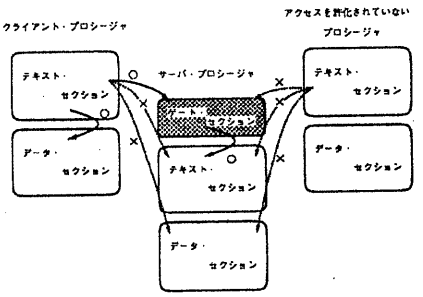

図 5: G セクションによる RPC の制御

3.2 ゲート・セクション

ゲート・セクション (G セクション) は Cubix に特有なメモリ・セクションで MULTICS おけるゲートと概念的には同一 (Organick[7]) である。G セクションはカーネルを介さずにユーザ・レベルで IPC を実行するために導入されたセクションで、セクション内にはその G セクションに対する分岐制御リストのみが格納されている。

クライアント・プログラムからサーバ・プログラムに jump する場合、直接にサーバ・プロシージャの T セクションに jump するのではなく、いったん、サーバ・プロシージャの G セクションに jump した後、G セクションから T セクションのエントリに jump する。このように G セクションを必ず経由することで、クライアント・プログラムからサーバ・プログラムのエントリ以外の番地に直接 jump できないよう、不正アクセスを防止している。

3.3 Unix 互換性

Unix オブジェクトは Cubix 空間の 4GB の境界ごとに Unix オブジェクトが配置される。Unix エミュレーション・モードではそれぞれのオブジェクトがゼロオリジンな 32 ビット Unix 空間にあるものとして実行される。64 ビット・プロセッサに上位 32 ビットがゼロとして扱われるようなアドレス・マスク機能が利用できれば、このような構成は比較的簡単に実現できる。

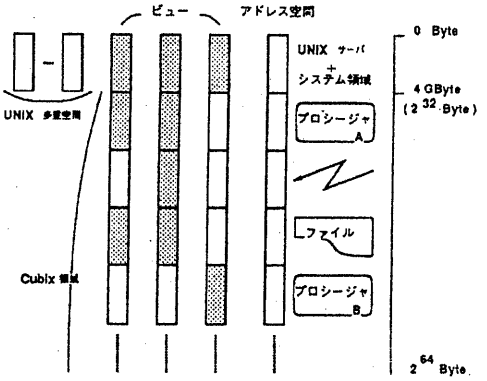

図 6: Unix 互換を実現するメモリ・モデル

また、さらに、上位 32 ビットに独立にアドレス・オリジンを設定できる機能がプロセッサに備わっていれば、32 ビットのアプリケーションを 64 ビット空間の自由な位置にロードすることが可能となる。

Cubix の Unix エミュレーションの特徴は以下の通り。

- Cubix マイクロカーネルおよび Unix サーバは 32 ビット Unix と同じアドレス領域にロードされる。
- 仮想空間の 0 から 4GB は Unix バイナリ・プログラムに解放し、移植された Unix アプリケーション・プログラムが動作する。
- Cubix プログラムはリンカが仮想アドレスを設定。同一仮想アドレスにロードされるプロシージャは唯一に定まる。また、Cubix の同一ビューに含まれるセクション間はコンテクスト・スイッチなしで IPC 可能である。
- 移植された Unix プログラムと Cubix プロシージャは Unix の IPC と同様、コンテクスト・スイッチを伴う IPC で通信する。

4 メモリ保護機構

Cubix は単一仮想空間のため、全てのプログラムやデータは同一の仮想空間上に存在し、また、入出力デバイスやタイマなどによる外部割り込みやスレッドの実行が終了しない限りコンテクスト切替えは行なわない。このようなアドレス空間のモデルにおいては IPC がカーネルを介さず、サブルーチンのように高速実行できる代わりに、

- アプリケーション・プログラムの不正アドレス・アクセスが検出不可能である。

といった欠陥が生ずる。そこで、Cubix では空間切替よりも粒度の細かいセクションまたはページ単位のメモリ保護機構を考えており、

- Unix がひとつの仮想空間にひとつのプロセスを割り当てる多重仮想記憶機構によって実現している不正アドレス・アクセス防止機能に相当するメモリ・セクションに関するアクセス制御。
- ファイルの open、close や IPC を行なうためのプロセス間のコネクションの確立などに相当する、あるスレッドが特定のメモリ・セクションにアクセス可能となるスレッドに関するアクセス制御。

を実現する。

4.1 OS インタフェース

Cubix のマイクロカーネルにおいては、プロシージャやファイルを実現するためのテキストに関するアクセス制御とファイル・アクセスや IPC の実行権をスレッドごとに付与するスレッドに関するアクセス制御と 2 種類のインタフェースを用意する。

4.1.1 テキストに関するアクセス制御

Cubix では複数のメモリ・セクションのアクセス関係を定義するサーバ層の概念として以下の 3 種を考える。

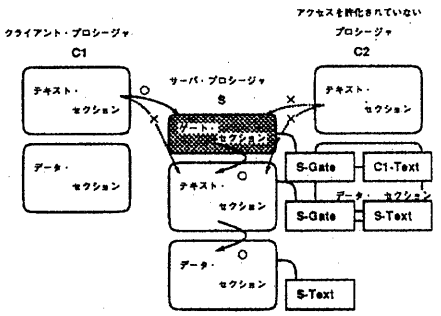

図 7: セクションのテキストに関するアクセス制御

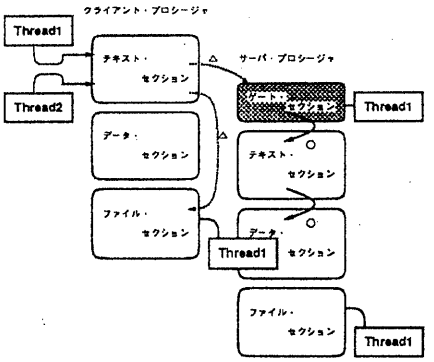

図 8: セクションのスレッドに関するアクセス制御

- プロシージャ データ・セクションおよびスタック・セクションにTセクションからのアクセスが許可される。

- ファイル クライアント・プロシージャからリクエストを受けたファイル・サーバはファイル・アクセス権をチェックし、許可する場合はクライアント・プロシージャのビューにDセクションをアタッチする。アタッチされたDセクションにはそのプロシージャのTセクションからのアクセス権が付けられる。

- IPC クライアント・プロシージャからのリクエストを受けたIPCサーバはコネクション権をチェックし、許可する場合はサーバ・プロシージャのGセクションにクライアント・プロシージャのTセクションのアクセス権を加える。IPCにおいてはGセクションに記述されている分岐条件にしたがってサーバ・プロシージャのTセクションにjumpするので、GセクションおよびTセクションにGセクションからのアクセス権が加えられている。

Cubixではプロシージャ、ファイル、IPCにおいて不正アドレス・アクセスを検出するためにTセクションおよびDセクションにアクセスを許可されたTセクションに関するアクセス制御リストが作成される。この制御リストによってアクセスを許可されたTセクション以外による不正なデータ・アクセスやIPCの分岐が検出される。

4.1.2 スレッドに関するアクセス制御

ファイル・アクセスは単に実行コードの内容のみに依存するのではなく、コンテクスト例えば入力パラメタなどにも依存する。つまり、図8においてたとえ同じプロシージャを実行していてもスレッド1はセクションFにアクセスできるが、スレッド2はFにアクセスできない場合があり、テキスト内にFへの不正アクセス・コードの存在の可能性を考えるとスレッド単位のアクセス制御が必要となる。

同様の制御はIPCにおいても必要である。クライアント・プロシージャからサーバ・プロシージャのGセクションへの不正アクセスに備えてIPCを許可されたスレッド識別子をGセクションに登録する。

4.2 ACLベースメモリ管理装置

Cubixでは前節に述べたように単一仮想空間における複数のプロシージャとファイルを格納し、高速RPCおよびデータベース・アクセスを実現するため、空間切替えよりも細かい粒度であるセクション単位のメモリ保護を行なうMMUを想定している。

ACLベースMMUは通常のページテーブル・エントリに加えてメモリ・セクションおよびスレッドに関するアクセス制御リスト(ACL)を管理する。ACLエントリはTLBにキャッシングされ、複数の比較器

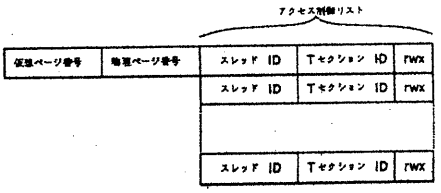

図 9: ACL ベース MMU のページテーブル・エントリ

により同時にアクセス権がチェックされる。そのため、TLB がヒットする限りアクセス制御チェックにともなう演算の遅延は生じない。一方、ACL フォールトが起こった場合はメモリ上のページテーブルがサーチされ、TLB エントリがスワップされる。

5 おわりに

本研究報告においては

- 64 ビット・アドレス空間を有効に利用し、
  - すべてのプロシージャを SVS に格納することによりサーバ・クライアント型アプリケーションを高速実行。
  - OLS による高速データアクセス。

  によりシステム性能の向上をはかる。
- スレッドとメモリ・セクションという単純なアブストラクションのみを提供し、多様な分散環境に適したマイクロカーネルによる実装により移植性の向上を目指す。
- Unix、Mach など標準 OS と互換。

といった特徴を有する 64 ビット指向 OS Cubix の特にカーネルに関する構想を述べた。

今後の開発予定としては

- i386 系のマシンで、セグメント・レジスタを利用して、Cubix のメモリ・モデルををエミュレートしながらマイクロカーネルを開発。
- ACL ベース MMU の詳細設計およびシミュレーションを行ない、64 ビット・マイクロプロセッサへ組み込む。

などを考えている。

参考文献

[1] Samuel Leffler, Marshall McKusick, Michael Karels and John Quarterman,"The Design and Implementation of the 4.3BSD Unix Operating System," *Addison-Wesley*, 1989.

[2] John R. Mashey,"64-bit Computing," *Byte*, pp.135–142, September 1991.

[3] Brian N. Bershad, Thomas Anderson, Edward Lazowska and Henry Levy,"User-Level Interprocess Communication for Shared Memory Multiprocessors," *ACM Transactions on Computer Systems*, Vol.9, No.2, May 1991, pp.175–198.

[4] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young,"Mach: A New Kernel Foundation For UNIX Development," *Proceedings of USENIX 1986 Summer COnference*, 1986, pp.93-112.

[5] Toshio Okamoto, Hideo Segawa, Sung Ho Shin, Hiroshi Nozue, Ken-ichi Maeda, Mitsuo Saito"A Micro Kernel Architecture for Next Generation Processors," *Proceedings of the Usenix workshop on MicroKernels*, 1992.

[6] Jeffrey C. Mogul and Anita Borg,"The Effect of Context Switches on Cache Performance," *ASPLOS-IV Proceedings*, April 1991, pp.75–85.

[7] Elliott Organick,"マルティクス・システム: システムのアーキテクチャとソフトウェア,"共立出版,1972.