

マイクロカーネル構成の OS におけるシステムサーバ

桑山 雅行[†] 最所 圭三[†] 福田 晃^{†‡}

[†]: 九州大学工学部

[‡]: 奈良先端科学技術大学院大学

マイクロカーネル構成 OS は、カーネルとして必要最小限の機能を提供するマイクロカーネルとそれ以外の OS として必要な機能を提供するシステムサーバから構成される。

このような構成のため、移植性や拡張性が良い、並列・分散環境との相性が良いといった長所がある。その反面、各モジュール間での通信オーバーヘッドが大きくそのため性能が劣るという欠点もある。

本論文では、システムサーバの分割数と、通信オーバーヘッドによる性能低下の関係について、実システム上に複数のサーバ構成の OS を構築し行った実験結果について述べる。

System Servers on Microkernel-based Operating System

Masayuki Kuwayama[†] Keizo Saisho[†] Akira Fukuda^{†‡}

[†]: Faculty of Engineering, Kyushu University

[‡]: Graduate School of Information Science

Advanced Institute of Science and TEchnology, Nara

kuwayama@csce.kyushu-u.ac.jp

A microkernel-based operating system consists of a microkernel and system servers. The microkernel provides the minimum functions. The system servers on the microkernel provide other functions.

Microkernel-based operating systems have advantages of portability, extendability and good compatibility on parallel/distributed environments etc., but also have a weak point of the communication overheads among each module. It is important problem to study theses merit and demerit.

We implement a few microkernel-based operating system, which have different system server architecture, and examine the relation of the number of system servers and communication overheads of those systems.

1 はじめに

マイクロカーネル構成オペレーティングシステム (OS) は、カーネルとして必要最小限の機能を提供するマイクロカーネルと、それ以外の OS として必要な機能を提供するシステムサーバから構成される。

マイクロカーネル構成 OS に関する研究課題としては、マイクロカーネルに関する研究課題、システムサーバに関する研究課題、システム全体の構成方式に関する研究課題など多くの研究課題があるが、本論文では、方式にいろいろなバリエーションが考えられシステムの性能に大きな影響を与える、システムサーバの構成方式に特に着目する。

現在までに研究・開発されてきたマイクロカーネル構成 OS では、主にシステムサーバは単一のサーバから構成されてきた。

しかし、システムサーバの移植性・拡張性などの向上や、並列計算機におけるハードウェアの並列性の活用のためには、システムサーバは複数のモジュールから構成されるべきであり、そのモジュール数は比較的大きくなる必要がある。

ところで、システムサーバを構成するモジュールの数が増えると、通信オーバーヘッドが増加しシステムの性能が低下することが懸念される。

本論文では、システムとして提供する機能をいくつかのシステムサーバにより実現するかというシステムサーバの分割数と、通信オーバーヘッドによる性能低下の関係について、単一プロセッサの実システム上に複数のサーバ構成の OS を構築し行った実験をもとに考察する。

2 マイクロカーネル構成の OS

2.1 構造

マイクロカーネル構成の OS は、カーネルとして必要最小限の機能を提供するマイクロカーネルと、それ以外の OS として必要な機能 (カーネル外機能と呼ぶ) を提供するシステムサーバから構成される。

システムサーバは一般にユーザ空間で実行される。

システムサーバはひとつだけで構成される場合もあるし、機能ごとにいくつかに分割されている場合もある[2]。

このように、OS 内部を複数のモジュールに分割した構成にすることにより、OS の移植性・拡張性などが高い。また、並列計算機の持つ並列性を活用しやすいという利点も持つ。

各モジュール (カーネル、各システムサーバ) は独立しており、直接他のモジュールのもつ資源にアクセスしたり、関数を呼び出したりすることはできない。

このため、システムサーバが他のシステムサーバやカーネルのサービスを要求する場合にはメッセージ通信が用いられる。また、ユーザがシステムコールを発行する場合にもメッセージ通信が用いられる。

メッセージ通信を行うためには、一度カーネルに処理を移す必要があるため、直接関数呼び出しを行う場合に比べ処理が重くなる。

マイクロカーネル構成 OS では、移植性・拡張性の向上や並列性の活用のためのモジュール化と、通信オーバーヘッドによる性能低下のトレードオフが重要な研究課題である。

2.2 本論文で取り組む研究課題

本論文では、上記のモジュール化と通信オーバーヘッドのトレードオフに関して、マイクロカーネル構成 OS の構成要素のうち、システムサーバの構成方式に関して特に着目して

議論する。

これは、システムサーバの構成方式は、マイクロカーネル自体の構成方式に比べ、方式に自由度が大きく、それによる性能の差が大きく現れると思われること、また、これまでに行われてきたマイクロカーネル構成 OS に関する研究^[1]は、単一サーバ構成のものが多く、システムサーバを複数のモジュールから構成する OS についてはまだあまり研究が進んでいないといった理由による。

システムサーバを単一のモジュールとして実現した OS から、システムサーバを多くのモジュールの集合として実現した OS までを実現し、それらの間で通信オーバーヘッドがどのように変化するかを見る。

3 MINIX オペレーティングシステム

本研究を行うに当たり用いる環境としては、MINIX オペレーティングシステム^{[3],[4]}を用いた。

MINIX は図 1 のように 4 つの層からなる構造をしている。第 1 層は、プロセスの多重化を司る部分であり、割り込み処理、メッセージ処理、コンテキストスイッチなどの処理を行う。

第 2 層は、主にデバイスの入出力を扱う部分である。この層は各々独立した、タスクと呼ばれるプロセス群からなる。タスクは他の多くのシステムではデバイスドライバと呼ばれるものである。第 1 層と第 2 層を合わせてカーネルと呼び、これらはシステム空間で実行される。

第 3 層は、システムサーバの層である。ここにはファイルシステム (FS) とメモリマネージャ (MM) の 2 つのシステムサーバが存在する。FS はファイルシステム全般の処理を行い、MM はメモリ管理やシグナル処理などを

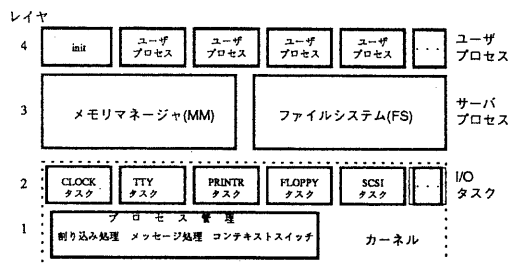


図 1: MINIX の構造

行うシステムサーバである。サーバはユーザ空間で実行される。

第 4 層には、一般のユーザプロセスが存在する。

このような構造をしているため MINIX はマイクロカーネル構成 OS の研究題材として適している。

4 システムサーバの実現

4.1 概要

今回の実験では MINIX のサーバを再構成し、次の 3 つのサーバ構成について比較評価を行った。

単一サーバ構成 FS と MM を 1 つにまとめたもの。

MINIX サーバ構成 MINIX のオリジナルサーバである FS と MM

細分割サーバ構成 FS と MM を 5 つのサーバに再構成したもの。

3 つの構成に関して、カーネルおよび実行するユーザプロセスに関しては、いくつかのテーブルを書き換えた程度であり、基本的に変更はない。

4.2 単一サーバ

MINIX サーバの FS と MM をひとつにまとめたサーバである。MINIX サーバで

FSとMMに分かれていた、すべてのシステムコールの処理を行う。

MINIX システムサーバでは、MM において、EXEC, EXIT, FORK, SETGID, SETUID, KILL の各システムコール処理の一部を FS へ依頼していた。そのためメッセージの送受信という重たい処理が必要であった。

しかし、単一システムサーバではこれが単なる関数呼び出しですむようになったためその分の性能向上が期待される。

4.3 細分割サーバ

ファイルシステムを管理する FS、メモリを管理する MM、ユーザおよびグループ ID を管理する XID、時間を管理する XTIME、シグナル機能を提供する XSIG の 5 つのサーバから構成したサーバである。XTIME は FS から、XID および XSIG は MM から分離させた。

このような分割により、例えばシグナル関係のところだけを変更したい場合に、MM 全体を再コンパイル・再リンクする必要はなくなり、XSIG システムサーバだけを変更し、再コンパイル・再リンクすればすむようになった。

しかし、例えば SIGNAL システムコールの処理は MM だけで行っていたのが MM、XID、XSIG の 3 システムサーバに分担されて処理されるようになり、システムサーバ間の通信量が増加するために性能が低下してしまうことが懸念される。

5 性能評価環境の構築

5.1 評価項目

以下の項目を各システムサーバ構成について測定する。

- システムコールの実行時間

あるシステムコールの実行にかかる時間を測定する。この時間が短ければ短いほど良いとする。

- システムサーバ間の通信回数・間隔

システムコールの実行中に、各システムサーバがどのシステムサーバとどのような間隔で何回通信を行うかを測定する。通信回数が少なければ少ないほど、また間隔が長ければ長いほど良いとする。このような記録を取ることで、システム内でどこがボトルネックとなっているかがわかる。

5.2 評価方法

5.2.1 システムコールの実行時間

システムコールの実行時間の測定には TIMES システムコールを用いる。TIMES システムコールは、システムコールを発行したプロセスに関する user time と system time、およびその子孫のプロセス群に関する user time と system time の 4 種類の時間を 1/60 秒単位で返すシステムコールである。

細分割方式のシステムサーバ構成の場合、他の 2 つのシステムサーバ構成の場合と比べて TIMES システムコール自体の処理時間に差があるのでその影響を除く必要がある。そのため今回の実験では、測定したいシステムコールを多数回繰り返し、TIMES システムコール自体の処理時間の差を無視できるようにしている。

TIMES システムコールを測定したいシステムコールの前後で実行しそれらの差をとることにより、測定したいシステムコールの実行に費やされた時間を求めることができる。

測定したシステムコールは、GETUID、SETUID のような単純なシステムコールと、FORK、EXEC のような複雑なシステムコールを測定し、システムコールの性質の違いに

```

if ( LOGFLAG が ON である. ) {
    送(受) 信者のプロセス番号を記録する.
    送信か受信か送受信かを記録する.
    メッセージのタイプを記録する.
    現在の時刻を記録する.

    記録を残すアドレスを増加させる.
}

```

図 2: 実行履歴の測定のために挿入されるコード

による差を見る。

また、NULL という何もしないシステムコールを作り、システムコールの処理におけるメッセージ通信の割合を見る。

5.2.2 システムサーバ間の通信回数・間隔の測定

システムサーバ間の通信回数・間隔は、システムコールの実行の際の実行履歴をとることにより測定する。実行履歴とは、例えば MM→XSIG→FS というように、ある処理の実行の際にどのシステムサーバが実行されたかの記録である。

実行履歴を測定する場合、計測によるプロンプト効果を最低限に押えるために計測の処理はできるだけ簡単にしなければならない。そこで、テストプログラム実行時にカーネル内のデータ領域に記録を残しておき、後で別のツールを用いてそれを見るという方法を用いる。

MINIX において、すべてのシステムコールはメッセージの送受信を用いて実現されている。そこで、カーネル内のメッセージの送受信を処理する部分に、誰が、誰に、どのようなメッセージを送信したか、およびそのときの時刻を記録するようなコードを挿入し記録を取る(図 2)。

すべてのメッセージの送受信を記録しても、必要のないところまで記録を残すことになる

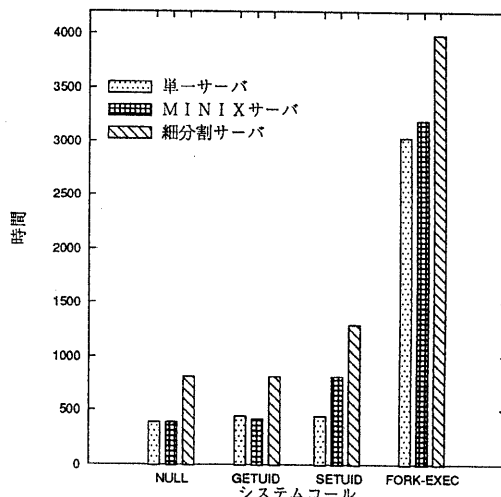


図 3: システムコールの実行時間

し、必要以上に記憶領域を用意する必要があるので、start_log(), end_log() といったルーチンを作り、必要な部分だけを記録するようにした。

start_log(), end_log() はそれぞれ、カーネルにメッセージを送り記録を取り始めることを知らせる、記録を取るのをやめることを知らせるルーチンである。

6 評価および考察

6.1 各システムコールにおける構成方式による違い

図 3 に、それぞれのシステムサーバ構成における各テストプログラムの実行結果を示す。NULL, GETUID, SETUID は、1 万回繰り返し実行したときの実行時間、FORK-EXEC は、FORK と EXEC を組み合わせて 500 回繰り返し実行したときの実行時間である。

NULL 細分割サーバ構成の実行時間が、単一サーバ構成と MINIX サーバ構成のおよそ 2 倍となっているが、これは NULL システムコールのメッセージを受けとった MM が、ユーザプロセスに

対して応答を返す場合に、XSIG が管理している変数の値を尋ねる必要があるためである。この場合、XSIG の行う処理は変数の値を返すだけなので、追加される時間はほぼメッセージの送受信のための時間である。このためおよそ 2 倍となっている。

GETUID GETUID システムコールの処理は各プロセスごとに用意されたデータ構造から ユーザ ID を読み出してその値を返すだけなので非常に簡単である。NULL の場合と同じ理由により、細分割方式サーバの実行時間が約 2 倍となっている。

SETUID MINIX サーバ構成 (細分割サーバ構成) では、ユーザ ID は MM(XID) で管理しているが、FS にもそのデータをキャッシングしている。そのために、ユーザ ID を設定した場合には、FS に対してその変更を知らせる必要がある。これに対して単一サーバ構成ではそのようなペナルティはない。これと NULL のところで述べた理由により、単一サーバ構成対 MINIX サーバ構成対細分割サーバ構成の比がおおよそ 1:2:3 となっている。

FORK-EXEC FORK,EXEC は非常に複雑な処理を行うシステムコールであり、そこで扱うオブジェクトは、ファイル、プロセス、メモリと多岐に渡っている。このような複雑な処理を行うシステムコールでは、必要な通信量も増えるが、その割合は相対的に小さくなる。これは、MINIX サーバ構成と、細分割サーバ構成で差があまりないことからわかる。

NULL と GETUID や SETUID とを比較すると、GETUID や SETUID として本当に必要な処理に比べ、メッセージの送受信の

ためのオーバーヘッドが非常に大きいことがわかる。

サーバとして実現するものはある程度その処理量が多いものでないと無駄が大き過ぎる。

このことから、本質的な処理が少ないものはマイクロカーネル内に入れた方が良いということが言える。

6.2 分割数による性能差

システムコールごとの処理時間の結果を見ると、単純なシステムコールでは、分割数による実行時間の差が 2~3 倍になることも珍しくないことがわかる。

一方、FORK,EXEC のような複雑なシステムコールでは、必要な通信回数は増加するが、そのオーバーヘッドが全体の実行時間に占める割合は相対的に小さいため、分割数による実行時間の差はそれほどない。

GETUID や SETUID などの単純なシステムコールと、FORK,EXEC のような複雑なシステムコールの間には処理に必要な時間が数十倍も異なる。このことを考えると、単純なシステムコールの実行時間に 2~3 倍の差が生じて、システム全体としての性能には差はあまり現れてこない。このことから、多少分割数を多くしてもそこでの通信オーバーヘッドによる性能低下は比較的小さく抑えられるのではないかと考える。

6.3 実行履歴

図 4 に、MINIX サーバ構成および細分割サーバ構成において、EXEC システムコールを実行した場合の実行履歴を示す。

図中において、各点はメッセージの送信が行われた時点である。

これより、システムサーバ間で頻繁にメッセージ通信が行われる様子がわかる。

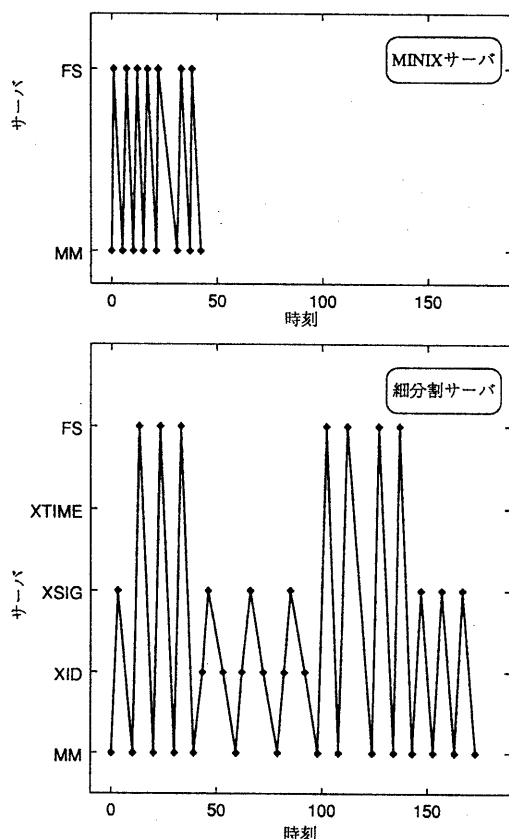


図 4: EXEC システムコールの実行履歴

ここで、MINIX サーバ構成と細分割サーバによって、MM がメッセージを送信する間隔が異っているが、これは次のような理由による。

もともと一つのモジュールであった MM を複数のモジュールに分割したが、それら分割されたモジュールの間でデータの相互参照によるデッドロックが生じないようにするために、MM のメインループの中でいくつかのチェックが行なわれている。このチェックのために、細分割サーバ構成では通信のオーバーヘッドに加え MINIX サーバ構成に比べさらに遅くなる。

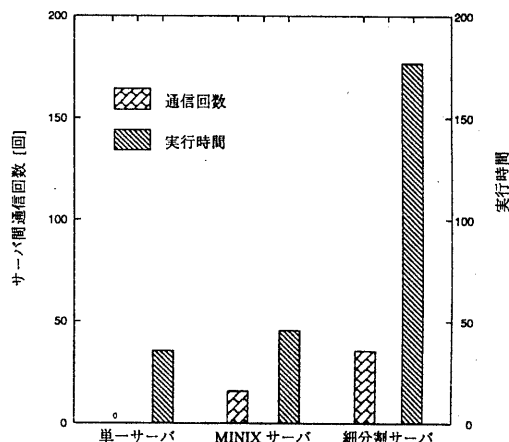


図 5: システムサーバ間通信回数と実行時間

6.4 システムサーバ間通信回数と時間

図 5 に、各システムサーバ構成において EXEC システムコールを実行した場合の、システムサーバ間通信回数と実行時間を示す。

これより、サーバ構成の変更に伴い、通信回数・実行時間ともに大きく増加していることがわかる。

ここでは通信回数の増加よりも実行時間の増加の方が著しい。これは上で述べた、複雑なシステムコールでは通信オーバーヘッドは相対的に小さくなるというのと矛盾するように見えるが、次のような理由による。

- 上で述べたデッドロックのチェックのための実行時間増加。
- 複雑な処理でも、EXEC のようにメモリ、ファイル、プロセスなど多くのものを扱う必要があるものは、通信の占める割合が多くなる。

6.5 細分割サーバ構成における各サーバ間の通信回数

図 7 に、細分割サーバ構成における各サーバ間の通信回数を示す。

	MM	XID	XSIG	XTIME	FS
MM		3	7	0	7
XID			3	0	0
XSIG				0	0
XTIME					0
FS					

図 6: 細分割サーバ構成における各サーバ間の通信回数

これより, EXEC システムコールにおいては MM ⇄ FS, MM ⇄ XSIG でのメッセージ通信が大きな割合を占めていることがわかる。

7 おわりに

システムサーバの分割数と通信オーバーヘッドによる性能低下の関係について, 実システム上に複数のサーバ構成の OS を構築し評価した。

各サーバ構成の OS の実現は, MINIX のサーバを修正して行った。

その結果, 単純なシステムコールではサーバ構成の違いによる差が大きいが, 複雑なシステムコールでは差はあまりでないことがわかった。単純なシステムコールはそもそも実行にかかる時間が短いので, サーバ構成により多少遅くなってもシステム全体としての性能の低下は小さいであろう。

現段階では, シングルプロセッサのパーソナルコンピュータ上での実験しか行っていない。今後は, マルチプロセッサシステムやマルチコンピュータシステム上でも, 並列処理による性能向上までも含めた評価を取りたいと考えている。

現在我々は, マルチプロセッサシステムを対象とした並列 OS K1^{[5],[6]} の研究・開発を行っている。今後, この研究での成果を K1 へ活かしていきたい。

参考文献

- [1] Golub, D., R. Dean, A. Forin, and R. Richard: "UNIX as an Application Program," Proceedings of the Summer 1990 USENIX Conference, pp.87-95
- [2] D.L.Black, D.B.Golub, D.P.Julin, R.F.Rashid, R.P.Draves, R.W.Dean, A.Forin, J.Barrera, H.Tokuda, G.Malan, and D.Bohman: Microkernel Operating Sstem Architecture and Mach, Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp.11-30(1992).
- [3] Andrew S. Tanenbaum: "OPERATING SYSTEMS : Design and Implementation," Prentice-Hall (1987)
- [4] Andrew S. Tanenbaum, Frans Meulenbroeks, Raymond Michiels, Jost Müller, Joseph Pickert, Steven Reiz, Johan W. Stevenson: "MINIX 1.5 REFERENCE MANUAL," Prentice-Hall (1991)
- [5] 今村信貴, 桑山雅行, 宮崎輝樹, 林茂昭, 福田晃, 富田真治: "並列オペレーティング・システム K 1 の設計と実現—フリー・プロセッサ・キューを用いたプロセッサ管理—," 並列処理シンポジウム J S P P ' 9 2, pp.305-312(1992).
- [6] 桑山雅行, 最所圭三, 福田晃: "並列オペレーティングシステム K 1—マイクロカーネルの考察と設計—," 情報処理学会システムシンポジウム, pp.69-76(1992).