システムソフトウェアと
オペレーティング・システム 65－1
（１９９４．７．２１）

マイクロカーネル構成OSにおける
システムサーバの構成法

桑山 雅行† 最所 圭三‡ 福田 晃‡

†九州大学大学院工学研究科
‡奈良先端科学技術大学院大学情報科学研究科

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として必要な機能を提供するシステムサーバから構成されるOSである．システムサーバをどのように構成するかはシステムの性能に大きな影響を与えるため，OSの設計の際に十分に検討されなければならない点である．システムサーバを構成するアクティビティとしてはプロセスを用いる場合とスレッドを用いる場合とが考えられる．我々はMINIXオペレーティングシステムを元にマルチプロセスサーバとマルチスレッドサーバを実装し比較を行った．その結果，マルチプロセスサーバでは入出力と他の処理のオーバラップ実行による性能向上が見られた．また，ユーザレベルスレッドを用いたマルチスレッドサーバではサーバの持つ並行性をうまく活用できないことを確認した．

Organization Schemes of System Servers in Microkernel–based Operating Systems

Masayuki Kuwayama† Keizo Saisho‡ Akira Fukuda‡

† Kyushu University
Hakozaki 6-10-1. Higashi–ku, Fukuoka 812, Japan
‡ Nara Institute of Science and Technology
Takayama-cho 8916–5, Ikoma–shi, Nara 630–01, Japan
E–Mail: kuwayama@csce.kyushu-u.ac.jp

A microkernel-based operating system consists of a microkernel and out-of-kernel functions implemented in user spaces. One of studies on these operating systems is for organization of the functions. It is crucial problem in concurrency, in particular in parallel/distributed environments. This is because organization and execution environment of the functions affects system performance. There are two methods to allow the activities in the out-of-kernel functions to be executed in parallel. One is multi-thread method. The other is multi-process method. There are trade-off between the multi-thread method and the multi-process one. There are few studies addressing the two methods. In this paper, we address this. We employ the MINIX operating system as the case study. Through implementing two system server organization according to the two methods, we compare them. Experiments show that with the multi-process server organization performance is improved due to overlaped execution of I/O and other processes . and that the multi-thread server organization using user level threads can't utilize parallelism which the server involves.

1 はじめに

マイクロカーネル構成のオペレーティングシステム (OS) は，マイクロカーネルとシステムサーバから構成される．マイクロカーネルは，割り込み処理，コンテキストスイッチ，メッセージ処理といったカーネルでしか実現できない最小限の機能を提供し，システムサーバは，さまざまな OS のエミュレーションや資源割当のポリシーなどの機能を提供する．

マイクロカーネル構成の OS は，その構成法としてさまざまなものが考えられる．これらの構成方法が OS の設計や性能に大きな影響を与える．特に，並列・分散環境においてシステムサーバが並行／並列に実行される場合にはその違いが大きく現れる．

システムサーバの構成の仕方としては，まず，サーバを構成するアクティビティに関して，単一のアクティビティで構成するか複数のアクティビティで構成するのかといった選択肢が考えられる．

サーバを複数のアクティビティで構成する場合にはさらに，アクティビティとしてプロセスを用いるのかスレッドを用いるのかといった選択肢が考えられる．

さらに，アクティビティとしてスレッドを用いる場合には，ユーザレベルスレッド／カーネルレベルスレッドのどちらを用いるのかという選択肢が考えられる．

また，サーバを複数のアクティビティで構成する場合，その数をいくつにするのか? システムサーバを分ける基準として何を用いるのか? といった点が問題となる．

システムサーバの数は，システムサーバ間の通信オーバヘッドを増減させる要因となる．システムサーバの数が多いとシステムサーバ間の通信が多くなり，数が少ないと通信は少なくなる．

一方で，システムサーバの数はシステムの持つ並行性に関係する．システムサーバを多くのアクティビティの集まりとして実現すれば，それらを並行して動作させることは容易であろう．逆に，システムサーバの数を少なくするとシステムに内在する並行性を引き出すことが面倒かもしれない．

システムサーバを何を基準として分けるのかについてはオブジェクトごとに分割する／メソッドごとに分割するといった選択肢が考えられるであろう[1]．

このようにいろいろな選択肢が考えられるが，それらが OS の設計法や性能に与える影響に関してはまだほとんど研究されていない[2],[3]．これらの与える影響を評価し，より良いマイクロカーネル構成 OS の構成を追求するのが本研究の目的である．

本論文では，アクティビティとしてプロセスを用いるシステムサーバとスレッドを用いるシステムサーバを単一プロセッサマシン上に実現し，それらを比較・評価することによりそれらがシステムの性能にどのような影響を与えるかを考察する．

本論文構成の構成は以下である．第 2 章では，本研究における実験環境である MINIX オペレーティングシステムについて簡単に述べる．第 3 章ではマルチプロセスサーバとマルチスレッドサーバを比較する．第 4 章，第 5 章ではそれぞれマルチスレッドサーバ，マルチプロセスサーバの実現について述べる．第 6 章で各サーバ構成のシステムの性能評価・比較について述べ，最後に第 7 章でまとめる．

2 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[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．

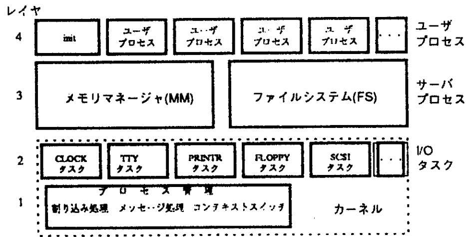

図 1: MINIX の構成

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．

3 マルチプロセスとマルチスレッド

アクティビティとしてプロセスを用いる場合，スレッドを用いる場合それぞれの利点・欠点を考察する．

アクティビティとしてスレッドを用いることには，以下のような利点がある．

1) 実現が容易である．多くのカーネル外機能は単一のプロセスとして提供されるので，プロセスを再構成することなくスレッド化したサーバを実現することができる．
2) スレッド間での通信オーバヘッドは小さい．これは，スレッドがひとつのアドレス空間を共有し共有変数を使って通信を行うことができるからである．

しかし，スレッドを用いることには以下のような欠点も存在する．

1) 高い性能を得ることができないことがある．実現の容易さの点から，マルチスレッド化サーバは単一の仮想プロセッサ上に実現されることが多い．この場合，その仮想プロセッサが I/O などのためにカーネル内でブロックするとサーバプロセス全体がブロックしてしまうため，他に実行可能なスレッドが存在してもそのスレッドを実行することができず性能が低下してしまう．

一方，アクティビティとしてプロセスを用いるサーバには以下のような利点がある．

1) それぞれのプロセスに仮想プロセッサが割り当てられるので，仮想プロセッサのブロックの問題が生じない．
2) 各サーバごとにアドレス空間を持つので，不正なアドレスへのアクセスからサーバを保護することができる．
3) さらに，この方式を研究することは OS の構造化に関する研究の一端を担うものである．

しかし，以下のような欠点も存在する．

1) OS を独立したモジュールに分割するのは非常に困難である．データの依存性を十分に考慮して分割しないと，それらの一貫性を保証するためのオーバヘッドが非常に大きくなる．
2) システムサーバ間での通信によるオーバヘッドが大きい[1]．
3) 使用するプロセスの数が増えると，必要な資源もそれだけ増加する．

上で述べたようにマルチスレッド方式とマルチプロセス方式の間にはトレードオフが存在する．しかし，これらの方式に関する実装を通した定量的な研究はこれまでほとんど行われていない．

本研究では，MINIX の FS をマルチスレッド化した．このマルチスレッド化した FS を，MTFS(Multi Threaded File System) と呼ぶ．また，FS をマルチプロセスも行った．このときのサーバ群を MPFS (Multi Process File System) と呼ぶ．次節，次々節においてそれぞれのサーバの実現について述べる．

4 マルチスレッドサーバの実現

スレッドの実現レベル　カーネルレベルスレッドは機能性があるが，カーネルが全ての処理を行う

ため処理が重たい．ユーザレベルスレッドはカーネルレベルスレッドに比べ非常に軽いという利点を持つが機能性に問題がある．例えば，あるユーザレベルスレッドが入出力等のためカーネル内でブロックすると他のユーザレベルスレッドがたとえ実行可能な状態であってもそれらを実行することはできない．このようなカーネルレベルスレッド，ユーザレベルスレッドの問題点に対処するため，カーネルとユーザレベルプログラムの協調によりスレッドを実現する方式も考えられている[5]．しかし，現状ではカーネルレベルスレッドや協調処理によるスレッドを提供するOSは少なく，また，それらの実現はユーザレベルスレッドの実現ほど容易ではない．我々はマルチスレッドサーバを実現するにあたり，まず第1ステップとして実現の容易なユーザレベルスレッドを用いた．

ユーザレベルスレッドライブラリ　表1にMTFSの実現に使用したユーザレベルスレッドライブラリのインタフェースを示す．このユーザレベルスレッドライブラリは我々の研究室で開発したものである [6]．

このライブラリでは，*th_fork*() がスタックエリアを確保するために *malloc*() を用いている．しかし，MINIX ではカーネル，システムサーバ，init プロセスがひとつのブートイメージにまとめられているため，システムサーバが *malloc*() を用いてヒープ領域からメモリを確保することができない．そのため，*malloc*() を用いずに，静的に割り当てられた領域をスタックエリアとして用いるようにした．

スレッドごとの変数　表2に各スレッドが保持すべき変数を示す．

スレッドの割り当て方策　機能をスレッドに割り当てる際には，以下のような選択肢が考えられる．

1) MTFS 内の全てのスレッドがサーバの全てのコードを実行可能とする．この方式は実行開始時にスレッドを生成することと，大域変数をロックするだけで実現できるという容易さを持つ．
2) 各システムコールに対応してスレッドを割り当てる．FS サーバはいくつかのシステムコールを提供している．サーバがシステムコールを受け付けると，そのシステムコールに対応したスレッドが実行される．この方式は，比較的簡単に実現可能である．しかしながら，各システムコールにそれぞれスレッドを割り当てると膨大な量のメモリがそのための消費されることになる．
3) スレッドをサーバのデータ構造に基づいて割り当てる．

実際の FS のデータ構造を調べた結果，FS の多くの機能がデータ構造の高範囲の部分を扱うためデータ構造をうまく分割することができず，方式 3) は実現が難しいことがわかった．また，方式 2) は，測定環境におけるセグメントサイズの制限 (64Kb) により多くのスレッドを生成できないため問題がある．そこで，我々は方式 1) を採用した．

MTFS サーバは3つの同じスレッドで構成される．それらのスレッドはサーバの全てのコードを実行することができる．これらのスレッドは，並行して動作することができるが，MINIX によって提供される仮想プロセッサは1つしか存在しないので並列に実行されることはない．

実行　MTFS 方式のサーバは以下のように実行される．サーバが起動されたとき，まず1つのスレッドが実行される．そのスレッドはいくつかの初期化を行った後，3つの子スレッドを生成する．その後，そのスレッドは子スレッドの終了を待つ (実際には子スレッドは終了することはないが)．各子スレッドは MTFS のメインループを実行する．メインループは，メッセージの受信，そのメッセージに対する処理，メッセージ送信者へのリプライという一連の処理の繰り返しである．子スレッドが実行を開始したときにはまずメッセージの到着を待つためブロックする．メッセージが届くとそのメッセージに応じた処理を行う．もしメッセージが I/O に関するものであれば，そのスレッドはそこで I/O 待ちのためブロックしてしま

表 1: ユーザレベルスレッドライブラリのユーザインタフェース

関数名	引数	意味
int th_fork(func, arg)	void (*func)(char *); char *arg;	関数 *func* を実行するスレッドを生成する．返り値は生成されたスレッドの ID である．
void th_exit(status)	int status;	スレッドを終了する．*status* は待ちスレッドへの返り値である．
int th_join(th)	int th;	スレッド *th* の終了を待つ．
void th_detach(th)	int th;	スレッド *th* をデタッチする．*th* 用に割り当てられたメモリはすぐに解放される．
void th_yield()		実行権を他のスレッドに渡す．
struct mutex * th_mutex_lock_init(mutex_id)	int mutex_id;	ロック変数を初期化し，その変数へのポインタを返す．
void th_mutex_lock(mutex)	struct mutex *mutex;	ロック変数 *mutex* をロックする．ロックを獲得できなかったら獲得するまでブロックする．
int th_mutex_trylock(mutex)	strunc mutex *mutex;	ロックの獲得を試みる．成功した場合非 0 を，失敗したら 0 を返す．
void th_mutex_unlock(mutex)	struct mutex *mutex;	ロック変数 *mutex* をアンロックする．

表 2: スレッドごとに持つ変数

変数名	意味
int who	呼びだし側のプロセス番号
int fs_call	システムコール番号
struct fproc *fp	呼びだし側の *fproc* 構造体へのポインタ
int super_user	スーパユーザかどうかのフラグ
int dont_reply	リプライ抑制のフラグ
int err_code	エラー番号の一時的格納場所
int rdwt_err	ディスク I/O リクエストの結果
message m	リクエストメッセージへのポインタ
message m1	リプライメッセージへのポインタ
off_t rdahedpos	先行読み込みする場所
struct inode *rdahed_inode	先行読みだしする i-node へのポインタ
char user_path[PATH_MAX]	ユーザパスネームの格納場所

う．もし，カーネルスレッドを用いれば別の FS　サーバに割り当てられたカーネルスレッドが他の

処理を続けることができる．本論文で用いているユーザレベルスレッドではスレッドがブロックするとMTFSサーバ全体がブロックしてしまう．

5 マルチプロセスサーバの実現

マルチプロセスサーバはMINIXのFSサーバを2つのサーバに分割することにより実現した．MINIXのFSサーバは，ファイルシステム関係の処理と時間管理の処理を行う．この2つの処理は独立しており，データも共有していないので2つのシステムサーバとして実現してもその間での通信オーバヘッドが問題になることはない．そこで，MINIXのFSサーバを，ファイルシステム関係の処理を行うNewFSサーバと，時間管理の処理を行うTIMEサーバとに分割した．これにより，MINIX FSサーバにおいては入出力の処理が行われている間は時間関係の処理は実行できなかったが，マルチプロセスサーバでは入出力待ちの間に時間関係の処理を行えるようになり，そのオーバラップ実行による性能向上が期待される．

MINIXサーバ構成，マルチスレッドサーバ構成，マルチプロセスサーバ構成において，メモリ管理処理を行うMMサーバの部分はいずれも同じである．

6 性能評価

MINIX FSサーバ，MPFSサーバ，MTFSサーバの性能を単一プロセッサマシン上で比較・評価した．

6.1 負荷

以下の2つの負荷に対する性能を評価した．

並行して実行される2つのプロセス *fileread*と*time*という2つのユーザプロセスを同時に実行する．*fileread*はファイルからデータを読み込むプログラムであり，*time*は，現在時刻を得るプログラムである．これらのプロセスの平均実行時間を測定する．MPFSサーバ環境ではこれらのプログラムはそれぞれNewFSおよびTIMEサーバに特に負荷をかける．そのため，この実験から，MPFSサーバ環境におけるNewFSサーバとTIMEサーバの実行のオーバラップの利点があきらかになる．

本論文で性能評価に用いた環境は，単一プロセッサのパーソナルコンピュータである．このため，各サーバが並列に実行されることはない．しかし，NewFSサーバの入出力待ちの間に他のサーバを実行することは可能である．

コマンド いくつかのコマンドを繰り返し実行する．ここで用いたコマンドは，*date*. *ls*, *make*, *pwd*, および*whoami*である．これらのコマンドの実行時間を測定する．

6.2 結果および考察

6.2.1 並行して実行される2つのプロセス

図2に上記の2プロセスを並行して実行させたときの実経過時間，システム時間，ユーザ時間を示す．各サーバ構成とも20回実行させた場合の平均値である．実経過時間は2つのプロセスが並行に実行開始されてから終了するまでに要した時間である．システム時間はプロセッサがカーネルおよびシステムサーバを実行していた時間である．ユーザ時間はユーザプロセス自身がプロセッサを使用していた時間である．

図2から，以下のことが言える．

システム時間 3方式の中でシステム時間が最も長いのはMPFS方式の場合である．MPFS方式ではサーバ間の通信オーバヘッドが大きくなることがあるが，ここで主に使われているNewFSとTIMEサーバの間では通信は行われないため，システム時間が長いのは通信オーバヘッドのためではない．MPFS方式においてシステム時間が長いのは，MINIXやMTFS方式に比べプロセスの数が多く，コンテキストスイッチが多く生じるためであると考えられる．

実経過時間 実経過時間を見てみると，MPFS方式が最も良い性能を示している．これは，MINIX

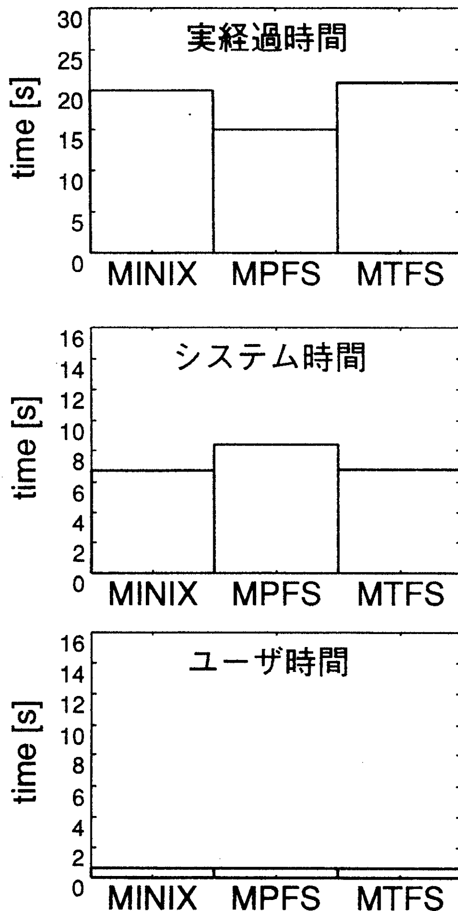

図 2: 並行して実行される 2 プロセスの実経過時間，システム時間，ユーザ時間

や MTFS 方式では *time* プロセスの実行時に FS サーバが入出力のためにブロックしていて *time* プロセスがブロックしてしまうことがあるが，MPFS 方式では入出力と時間関係を扱うサーバがそれぞれ NewFS と TIME サーバに分かれているため，*time* プロセスは入出力が行われている間にブロックすることなく実行を続けることができるからである．

ユーザ時間　ユーザ時間に関してはサーバの構成による違いはない．これは，サーバの構成方式の違いはシステムコールの実行や，割り込み処理といった OS 内部での処理にのみ関係し，ユーザの記述したプログラム部分の実行には影響しないからである．

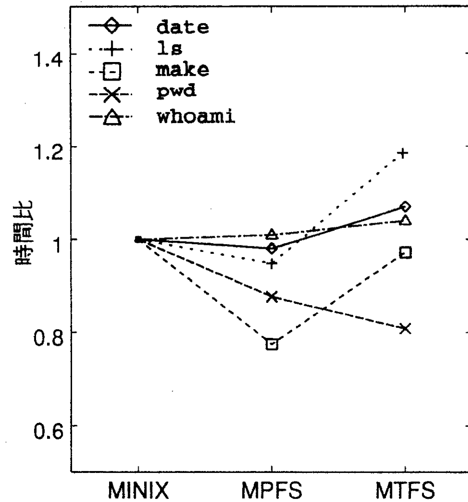

図 3: コマンドの実行時間

MTFS　MTFS 方式は MINIX とほぼ同じ性能を示した．これは，この実験を単一プロセッサ上で行ったこと，用いているスレッドがユーザレベルスレッドであることによる．マルチプロセッサマシンならばサーバに複数の仮想プロセッサを与えることにより複数のスレッドを並列に実行し，より高い性能を得ることが可能である．

6.2.2　コマンド

図 3に，いくつかのコマンドの実行時間を示す．

図 3から以下のようなことが分かる．*whoami* コマンドでは 3 方式に違いがほとんど見られない．これは，このコマンドがファイルシステムと無関係であるためである．

date コマンドでも方式による違いは見られない．これは，*date* コマンドは FS サーバ (MPFS 方式の場合は TIME サーバ) を使うが，処理が簡単でそれらのサーバに負荷をあまりかけないためであろう．

make コマンドの実行時間は MPFS 方式がかなり短い．これは，*make* コマンドが多くの I/O を行うと同時に，time や utime といったシステムコールを多く発行するためである．これは，MPFS 方式の利点を表している．

実行するアプリケーションが計算ばかりを行うようようなものならばOSの内部構造の違いはほとんど影響が無いが，アプリケーションが入出力を多く行うものならば，ユーザレベルスレッドを用いたマルチスレッドサーバよりもマルチプロセスサーバの方が良いと言える．

全体としてMPFS方式が良い性能を示した．しかし，もしMTFS方式のスレッドとしてユーザレベルスレッドではなくカーネルレベルスレッドを用いたならば，MTFS方式はもっと良い性能を示すであろう．なぜならば，カーネルレベルスレッドを用いればサーバ内のスレッドがカーネル内でブロックした場合でも，ユーザレベルスレッドを用いた場合のようにサーバ全体がブロックしてしまうのではなく，サーバは別のスレッドにより別の処理を続けることができるからである．

7 おわりに

マイクロカーネル構成のOSにおいて，システムサーバの構成は通信オーバヘッドやシステムの並行性など，OSの設計や性能に大きな影響を与えるものである．本論文では，システムサーバを構成するアクティビティとしてプロセスを用いた場合とスレッドを用いた場合について比較・評価を行った．いくつかの実験の結果，プロセスを用いた場合には入出力と他の処理のオーバラップ実行により性能の向上が見られたが，ユーザレベルスレッドを用いた場合には，仮想プロセッサがカーネル内でブロックするとシステムサーバ全体がブロックしてしまい，システムサーバの持つ並行性を活かすことができなかった．

ユーザレベルスレッドではなく，カーネルレベルスレッドやカーネルとユーザレベルプログラムが協調するようなスレッドを用いればこの問題に対処することができる．このようなスレッドを用いたサーバの実現・評価が今後の課題である．また，今回の実験環境は単一プロセッサマシンであったが，並列処理環境であればシステムサーバの並列実行によりいっそうの性能向上が期待される．並列処理環境における構成法の違いの研究も今後の課題である．

参考文献

[1] M.Kuwayama, K.Saisho and A.FUkuda: A scheme for Organization of System Servers in Microkernel-based Operating System and Its Performance, Proc. Joint Conf. on Software Engineering, pp.193-199(1993).

[2] Black, D.L., et al.: Microkernel Operating System Architecture and Mach, Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp.11-30(1992).

[3] Julin, D.P., et al.: Generalized Emulation Services for Mach 3.0 — Overview, Experiences, and Current Status, Proc. the USENIX Mach Symp., pp. 13–26(1991).

[4] A.S.Tanenbaum: Operating Systems: Design and Implementations, Prentice-Hall(1987).

[5] Anderson, E., et al.: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", Proc. the 13th ACM Symp. on Operating Systems Principles., pp. 95–109(1991).

[6] T.Miyazaki, M.Kuwayama, K.Saisho and A.Fukuda: Parallel Pthread Library (PPL): User-level Thread Library with Parallelism and Portability, COMPSAC'94(1994).