動的保護が可能な動的構築機構を有するオペレーティングシステム・サーバの実現と評価

柏木一彦† 福田晃† 最所圭三†

本論文では，計算機ハードウェア上に直に実装されたオペレーティングシステム（ベースOSと呼ぶ）上で，ユーザにベースOSとは異なった仮想計算機環境を与える応用プログラム（OSサーバと呼ぶ）を対象として，動的保護が可能な動的構築機構を有するOSサーバを実現し，その性能を評価する．本論文で対象とする環境は次のとおりである．(i) OSサーバは，小さなカーネルと，カーネルから処理を依頼される複数のサーバタスクから構成される．(ii) OSサーバと，それを利用する複数のユーザタスクは，ベースOSが提供する1つの仮想アドレス空間内でのみ動作している．(iii) 当該仮想アドレス空間は，カーネル空間およびユーザ空間と呼ぶ2種類の空間から構成されており，カーネルはカーネル空間に，サーバタスクとユーザタスクはユーザ空間にあり，各々静的固定的に保護されている．本論文では，このOSサーバに拡張を加え，(i) サーバタスクのカーネル空間への動的な取り込み/追い出し，(ii) 新たな機能の動的追加/取り込み/追い出し/置換，(iii) カーネルスケジューラの動的置換，が行えるとともに動的保護が可能な動的構築機能を有するOSサーバを実現した．さらに，性能評価を行い，(i) サーバタスクの取り込みによるシステムコールの実行時間の短縮，(ii) 動的保護のオーバヘッド，(iii) カーネルスケジューラの動的置換の効果，などを定量的に明らかにした．

Implementation and Evaluation of a Dynamically Reconfigurable Operating System Server with Dynamic Protection

KAZUHIKO KASHIWAGI,† AKIRA FUKUDA† and KEIZO SAISHO†

In this paper, we implement and evaluate a dynamically reconfigurable operating system server with dynamic protection. The server is an application program running on a base operating system. The server we consider in this paper is; (i) the server consists of a small kernel and server tasks, (ii) the operating system server and user tasks run in a single virtual address space supported by the base operating system, and (iii) the virtual address space consists of two kind of spaces, called kernel space for the kernel and user space for the server tasks and user tasks, which spaces are statically protected. By extending this operating system server, this paper implements a dynamically reconfigurable operating system server with dynamic protection which supports; (i) dynamic injection (removing) of the server tasks into (from) the kernel space, (ii) dynamic adding, injection, removing, and exchanging of new functions, and (iii) dynamic exchanging of a kernel scheduler. The results of performance evaluation show; (i) the dynamic injection of the server tasks improves system performance, (ii) the dynamic protection is relatively large, and (iii) the dynamic exchanging of the kernel scheduler provides well-suited environments for user tasks.

1. はじめに

社会の多様化にともない，応用プログラムが計算機環境に対して望むサービス，すなわちオペレーティングシステム（以下OSと示す）に対して望むサービスが多様化している．たとえば，マルチキャスト通信を利用したい応用プログラムをあるOS上で実行させたい場合，当該OSがそれを提供していなければ当該OSを変更する必要がある．このとき，マルチキャスト通信を提供できるようにOSコードを書き直して再度OSを立ち上げれよいが，立ち上げ後に，今度はブロードキャスト通信を望む応用プログラムを実行させたい場合には，OSの変更と立ち上げが再度必要となる．このOSコードの書き直し，再立ち上げには多大な労力を必要とする．また，多様化している応用プログラムの中で，応用プログラムがOSに望むサービスをあらかじめ予想して，OSプログラムを作成する

† 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

ことは困難である．さらに，複数の応用プログラムがOSが提供している同じ機能を利用する場合でも，当該応用プログラムごとに，当該機能の実現方法を変えた方がよいこともある．たとえば，応用プログラムごとに実行パターンが異なれば，そのスケジューリング方式を変えた方がよい．このような状況に対処するための鍵となる概念は，OS実行中に，動的に機能を追加したり，追加した機能を交換したり，さらには，OS自身があらかじめ提供している機能を動的に変更できる，すなわち動的構築機構の提供である．従来の古典的なOSは，機能，機構があらかじめ固定された設計概念に基づいており，動的構築機構を有するOSの設計，開発が求められている．本論文は，この動的構築機構を有するOSに関するものである．

動的構築機構を有するOSに関しては，(i) OSアーキテクチャ，(ii) OSの実装レベルに関していくつかの選択肢がある．以下にこれらについて考察する．

(1) OSアーキテクチャ

OSプログラムをどのように構成するかという構成方法であり，以下の2つの構成方法がある．

(i) 単層カーネルアーキテクチャ

OS自体を1つの巨大なプログラムとして作り，プログラム内の各機能を関数呼び出しで構成する方法であり，OS自体が高速に実行できるという利点を有する反面，OS自体の修正・変更などが困難となる．

(ii) マイクロカーネルアーキテクチャ

OSとしての必要最小限の機能をマイクロカーネルとして提供し，その他の機能をその上で動作する複数のサーバタスク（たとえば，ネットワーク管理，高レベルのメモリ管理，ファイルシステムなど）として提供する方法である．マイクロカーネルは，必要に応じてサーバタスクにメッセージ通信を用いて処理を依頼することにより，応用プログラムからの要求を処理することになる．また，本方法の実現方式を保護の観点からみると，プログラムを実行する空間として，互いに保護されたカーネル空間とユーザ空間と呼ばれる2種類の空間を設け，マイクロカーネルはカーネル空間内で，サーバタスクはユーザ空間内で実行される方式を採用する場合が多い．マイクロカーネルアーキテクチャでは，OSの修正がサーバタスクの修正で済むことが多く，また，新たな機能を追加する場合には，当該機能を新たなサーバタスクとして実現すればよいなど，動的構築機能を有するOSのアーキテクチャに適しているため，本論文では，マイクロカーネルアーキテクチャを採用する．しかし，マイクロカーネルとサーバタスクとの間のメッセージ通信のオーバヘッドにより，OS自体の性能が，上記(i)よりも劣るという欠点があるので，これを改善する必要がある．

(2) 実装レベル

動的構築機構を提供するOSの実装レベルとしては，(i) 計算機ハードウェア上に直に実装するレベル，すなわち従来の古典的なOSが実装されたレベル，または，(ii) 計算機ハードウェア上で直に実装されている既存のOS（本論文では，以後，ベースOSと呼ぶ）の上で，ベースOSとは異なる仮想計算機環境をユーザに提供する1つの応用プログラム（本論文では，この応用プログラムを以後，OSサーバと呼ぶ）として実装するレベル，の2つの方法がある．上記(i)の方法は，計算機ハードウェア上に直に実装するので，OS自体が高速になる，ハードウェアが提供する機能を直に利用できるので，きめ細かい動構築機構を提供できる，という利点がある反面，いったん動的構築機構を有するOSを構築した後に，その動的構築機構の仕様を変更する場合には，多大な労力を必要とするという欠点がある．一方，上記の(ii)の方法では，(i)とは逆の利点，欠点を有することはもとより，現在，ユーザが要求する仮想計算機環境が多様化しているため，1つのベースOS上に複数の異なる計算機環境を提供するため，複数のOSサーバを動作させることが行われており，この傾向は今後ますます強くなると考える．そこで，動的構築の対象を，ベースOSではなく，OSサーバとすることは，意義のあることである．したがって，本論文では，OSサーバとして実装する．

本論文では，構成方式の観点からはマイクロカーネルアーキテクチャ，実装レベルの観点からはOSサーバ方式を採用し，動的構築機構に関しては，以下に示す機構を対象とする．

(i) マイクロカーネルの欠点を改善する機構：サーバタスクをカーネル空間へ動的に取り込む機構を提供し，これにより，マイクロカーネルとサーバタスクとの間のメッセージ通信のオーバヘッドを削減する．また，取り込まれたサーバタスクを動的にカーネル空間からユーザ空間へ追い出す機構も提供する．本機構は，サーバタスクを置換する際に必要な機構である．なお，サーバタスクの置換については，その実現にあたっては困難で複雑な課題が存在し，本論文の範囲を超えるので今後の課題とする．

(ii) 新機能の動的追加/取り込み/追い出し/置換を行う機構：OSに新機能を新たなサーバタスクとして追加したり，さらには，メッセージ通信のオーバヘッドを軽減するためにカーネル空間へ取り込んだり，また，置換するために，カーネル空間からユーザ空間に

追い出したり，さらに新たな機能と置換する機構を提供する．

(iii) カーネル内スケジューラの動的置換機構：あらかじめ組み込まれているカーネル内機能を動的に置換する．この機能の中で，本論文では，最も要望があると思えるスケジューラを対象として，動的に異なるスケジューラに置換する．

(iv) 動的保護機構の提供：上記における，サーバタスクおよび新機能のカーネル空間への取り込み，またはカーネル空間からの追い出しのために，動的保護機構の提供が必要となる．

OSの動的構築に関する研究・開発に関しては，対象をベースOSとしたものはいくつか行われているが[1)~7)]，OSサーバを対象とした研究は行われていない．あえて関連した研究を述べれば，文献8)がある．しかし，文献8)は，応用プログラム内の一部分を動的に入れ換える研究であり，また，応用プログラム内のコンテクスト（スレッド）の数は1つであり，本論文が対象とする環境（マルチスレッド環境）および上記(i)~(iv)の機構を対象としたものではない．

本論文では，OSサーバに関してあるモデルを仮定し，このOSサーバを拡張して，上記機構を有するOSサーバを設計・実装し，上記機構の有効性を定量的に評価することを目的とする．

本論文の構成を述べると，仮定するOSサーバモデルを述べ，上記(i)~(iv)の機構を実現するための検討課題・選択肢などを整理するとともに考察を加え，本論文で採用するアプローチを述べる．次に，その実現方法の概要を述べ，MINIXサーバを拡張する方向でOSサーバを実装し，その性能評価を行う．

2. 対象とするOSサーバのモデル

拡張の対象とするOSサーバのモデルには，種々のモデルが考えられるが，本論文では，**図1**に示す以下のものに限定する．

(1) 動作モデル

(i) 対象とするOSサーバは，単一プロセッサ構成の計算機上のベースOS（本論文ではUNIX系とする）上で動く．

(ii) ベースOSからみれば，OSサーバとその上で動作するユーザタスクは，全体で1つの仮想アドレス空間を構成し，当該アドレス空間はベースOSが提供する仮想プロセッサを1つだけ用いて実行される．すなわち，当該仮想アドレス空間を実行するためにベースOSが割り当てているコンテクストは，1つのみである．

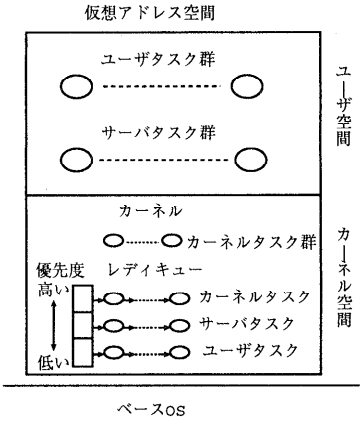


図1 OSサーバモデル
Fig. 1 The operating system server model.

(2) OSサーバ

(i) OSサーバは，1章で述べたようにマイクロカーネルアーキテクチャをとる．すなわち，マイクロカーネルと，その上で動くいくつかのサーバタスクがある．マイクロカーネル（以後，単にカーネルと呼ぶ）内には複数のカーネルタスクが存在する．また，カーネルタスク群，サーバタスク群，ユーザタスク群は，各々コンテクストを持ち，カーネルの基本的部分でこれらを管理する．したがって，カーネルは，ベースOSからみれば，ユーザレベルのマルチスレッド機能をサポートしていることになる．

(ii) ユーザタスクからのシステムコールは，ベースOSへのソフトウェア割り込みを介して，カーネルに制御が移る．タスク間通信はメッセージ通信である．

(iii) 入出力処理は，最終的にはベースOSへシステムコールの形で処理を依頼する．

(3) スケジューリング

(i) 基本的には優先度スケジューリングで，カーネルタスク，サーバタスク，ユーザタスクの3つの優先度がある．優先度は，カーネルタスクが最も高く，次にサーバタスク，ユーザタスクの順である．

(ii) カーネルタスク群，およびサーバタスク群は各々FIFOでスケジューリングされる．

(iii) ユーザタスク群はラウンドロビン（以下RRと示す）でスケジューリングされる．

(4) 保護

当該仮想アドレス空間は，カーネル空間とユーザ空間と呼ぶ2つの空間からなっている．カーネルはカーネル空間内で実行され，サーバタスクとユーザタスクはユーザ空間でのみ実行される．各々の空間は互いに保護されている．すなわち，各々の空間に制御が移る

と，他空間へはアクセス不可となるように，ベースOSへシステムコール（本論文では具体的にはmprotectシステムコールである）を発行して保護される．ここで，留意してほしいのは，この保護モデルでは，ユーザタスクおよびサーバタスクをカーネル空間内で実行する機構が提供されていないことである．本論文では，この機構（動的保護機構）も実現する．

上記のモデルは，単一仮想アドレス空間内で動作するマイクロカーネルベースのOSサーバとしては，特に特化したものではなく，比較的一般的なモデルと考えられる．

3. 検討事項

本章では，2章で仮定したOSサーバモデルの下で1章で述べた動的機構を実現するための検討課題を整理・考察し，議論する．

3.1 保護と安全性

(1) 保護の実現方法

保護に関しては，カーネル空間に取り込んだサーバタスクおよび新機能の保護モデルに関して以下の2つが考えられる．

(a) カーネル空間の拡張と見なす方法：取り込んだサーバタスクおよび新機能は，カーネル空間が拡張されたと見なし，いままでのカーネルを保護する場合と同等に扱う方法である．

(b) 従来のカーネル空間とは独立させて保護する方法：取り込んだサーバタスクおよび新機能を，(a)とは異なり，当該サーバタスクまたは新機能が呼び出されたときに，保護する方法である．

上記(a)は，従来のカーネル空間と同等に扱うという一貫した見方ができるという利点がある反面，取り込んだサーバタスクおよび新機能が，従来のカーネル空間にアクセスできることになるので，安全性が損なわれる欠点も有する．一方，上記(b)はその逆となる．本論文では，カーネル空間の拡張と見なす素直な考え方に重点をおいて，方法(a)を採用することにする．

また，取り込み対象のタスクは，仮想アドレス空間内の一部の領域に配置されているが，取り込まれたとき，この配置を変更するか否かの選択肢がある．すなわち，**図2**に示す以下の配置方法がある．

(a) 再配置方式：図2(a)に示すように，カーネルが配置されている領域に連続するよう再配置する方法である．

(b) 固定方式：図2(b)に示すように，取り込まれたタスクの配置は変えない方式である．この方法では，カーネル空間は不連続な領域となる．

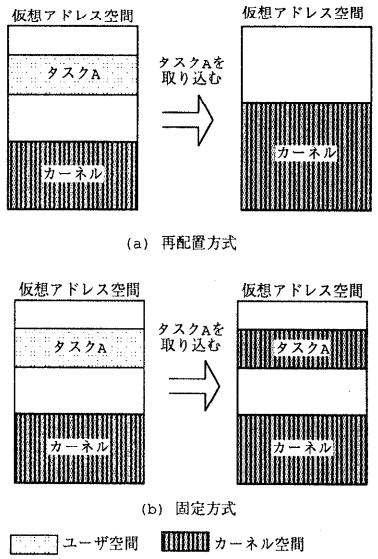


図2 取り込み対象タスクの取り込み後の配置方法
Fig. 2 Allocation method of address space of a new task after injection.

上記(a)は，カーネル空間が連続した1つの領域から構成されるので，カーネル空間を保護する場合に，ベースOSへのmprotectシステムコールの発行回数を少なくできるという利点を有するが，取り込み対象のタスクのアドレスを解決しなければならない．一方，上記(b)の利点・欠点は，(a)の逆となる．本論文は，(b)の利点を優先して，方法(b)を採用する．

(2) 安全性

取り込まれた新機能のカーネルに対する安全性はきわめて重要な問題であるが，究極的にはその意味をも考慮しなければならない複雑な問題となり，本論文の対象範囲を超えるので，本論文では不完全ではあるが，きわめて単純な方法を採用する．すなわち，動的構築のためのシステムコールが発行できるのは，ルート権限のあるユーザタスクのみに制限する．すなわち，安全性については，ルート権限のあるユーザの責任に委ねることにする．安全性に対するその他の対策は，今後の検討課題とする．

3.2 サーバタスクの取り込みおよび追い出し

3.2.1 取り込み

(1) 取り込み後の実行モデル

取り込んだサーバタスクを新たなカーネルタスクとするのか，既存のカーネルタスクに取り込むかの選択肢がある．すなわち，以下の選択肢がある．

(a) 新たなカーネルタスクとする方法：取り込んだサーバタスクを新たなカーネルタスクとし，そのコンテクストを引き継ぐ方法である．本方法の利点は，取

り込んだサーバタスクをカーネルタスクとして独立させるので，当該サーバタスクの取り込み直前のコンテクストをそのまま引き継げるなど実現が容易になるなどの利点がある．その一方で，カーネルタスク間通信がメッセージ通信となるので，メッセージ通信のオーバヘッドが生じる欠点がある．この欠点を改善するために，1章で述べたように，他のカーネルタスク内の一部を当該カーネルタスクに取り込んで，当該カーネルタスクからのメッセージ通信を一部関数呼び出しにして，オーバヘッドを削減する方法をとる．

(b) 既存のカーネルタスク内に取り込む方法：対象サーバタスクを，カーネル内の既存タスクに取り込む方法である．本方法は，メッセージの受け口の変更など，大幅な変更を必要とし，実現が上記 (a) よりも困難となることが考えられる．

本論文では，実現の容易性とサーバタスクの実行時間がメッセージ通信に要する時間よりも大きいと考えられることより，方法 (a) を採用する．また，前述したように，他カーネルタスク内の一部を当該サーバタスクに取り込むことにより，カーネルタスクとの間のメッセージ通信のオーバヘッドの削減を試みる．なお，本論文が対象とするOSサーバモデルの仮定より，このように行っても正常に動作する．

サーバタスクの取り込み前と取り込み後での，サーバタスクの実行の様子を理解するために，**図3** の例を用いて説明する．図3は，ある既存システムコールをサーバタスクAが処理するとして，その処理中で，カーネルタスクaが有する機能（関数fa）を利用する際に，サーバタスクAがユーザ空間で動作する場合（図3(a)）と，カーネル空間に取り込まれている場合（図3(b)）の当該システムコールの処理の流れを示したものである．図中，(1) の通信方法が，(a) ではメッセージ通信であるのに対し，(b) では関数呼び出しにすることにより通信のオーバヘッドを軽減している．

(2) 取り込み時期

取り込みは，ユーザタスクからのシステムコールにより行う．取り込み時期は，取り込み対象のサーバタスクが使用中でないときである．この状況を考えてみる．当該サーバの状態としては，(a) 実行中，(b) 他からのメッセージを受け取って，実行可能状態，(c) 処理中に他サーバにメッセージを送ってその返り値を待つサスペンド状態，さらに，(d) アイドルでサスペンド状態がある．ユーザタスクからの取り込みシステムコールがいつ実行されるかを上記の場合について考える．ユーザタスクの優先度はサーバタスクよりも低いので，サーバタスクが実行中または実行可能状態のときは，当該ユーザタスクはスケジュールされない．また，入出力関係の処理は，ベースOSへのシステムコールとなり，このシステムコールはOSサーバからみれば同期通信となる．すなわち，ベースOSへシステムコールを発すると，当該仮想アドレス空間に割り当てられているベースOSの仮想プロセッサがブロックされ，システムコールが返ってくると，中断したコンテクストから処理が再開される．また，(c) の場合では，送り先は他のサーバタスクまたはカーネルなので，当該ユーザタスクが実行以前にメッセージで依頼した処理が返ってきていることになる．さらに，RRの実現のためのベースOSからのタイマ割り込み（ソフトウェア割り込み）によるコンテクストスイッチングは，ユーザタスクのみにしか適用されない．

したがって，取り込みシステムコールをユーザタスクが発行したときには，サーバタスクはアイドル状態であるので，従来行われているような使用中か否かを示すフラグを設ける必要がないことが分かる．

なお，上記の結論が成立するためのOSサーバモデルの仮定をまとめてみると，用いた仮定は2章で述べた仮定の中で，(1)(ii)，(2)(iii)，(3)(i)，(ii) である．これらの仮定が成立しない環境，たとえば，ベースOSが提供する複数のスレッドを用いてOSサーバが実装されている場合（この場合は，2章の (1)，(ii) が不成立）などでは上記の結論とはならないことに留意され

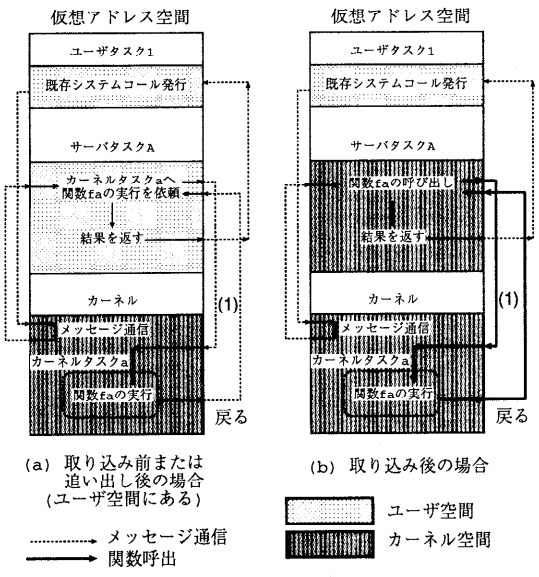


図3 既存システムコールの処理の流れ（サーバタスクAを介する処理）

Fig. 3 Execution flow of an existing system call through the server task "A" before/after injection.

たい．

3.2.2 追い出し

ユーザ空間への追い出しも，取り込みと同様にユーザタスクからのシステムコールの形で行う．

(1) 追い出し時期

追い出し時期は，当該サーバタスクが使用されていないときである．この場合を考えると，上記3.2.1項で考察したように，追い出しシステムコールを発行したときには，当該サーバタスクはアイドルでサスペンドしている場合である．したがって，取り込み操作同様に，使用中か否かを示すフラグは必要なくなる．

(2) 追い出し後の実行モデル

ユーザ空間に追い出されたサーバタスクは，追い出し直前のコンテクストを引き継いで，1つのタスクとして動作する．また，このような実行モデルでも，OSサーバモデルの仮定より正常に動作する．

3.3 新機能の追加/取り込み/追い出し/置換/実行

ユーザタスクからのシステムコールの形で行う．

(1) 新機能の追加（登録）

新機能を含んだ処理をユーザタスクとして立ち上げ，登録システムコールにより，カーネルに認識させることにする．4.1節にも後述するように，登録システムコールにより，登録した新機能は，ユーザが利用できる（後で置換する目的で，置換先機能を登録する場合はこの限りではない（後述の4.1節(2)を参照されたい））．このとき，登録した機能はユーザ空間で動作する．当該ユーザタスクは，カーネルとのメッセージ送受信機能を有し，登録された後は，メッセージ待ちでサスペンドしておくように，プログラミングしておく必要がある．新機能で許す処理内容は，カーネル内データを読み込むだけの処理内容とする．

(2) 新機能の取り込み

新機能をカーネル空間に取り込むにあたっては，取り込みシステムコールの発行に先立って，登録システムコールを用いて当該新機能を登録しておく必要がある．

(i) 取り込み時期

取り込み時期は，使用中でないときである．前述したサーバタスクの場合とは異なり，新機能はユーザタスクの一部として動作し，ユーザがその機能を利用できるので，取り込みシステムコールが発行された時点では，取り込み対象のユーザタスクは使用中である可能性がある．したがって，これを判別するため，使用中か否かを示すフラグが必要となる．すなわち，ユーザタスク内でフラグを設け，初期値を0にし，当該新機能に入るときにフラグを+1，出るときに−1にし，フラグの値が0であれば，当該新機能が使用中でないことになる．

(ii) 取り込み後の実行モデル

ユーザタスクとして動作させている新機能をカーネル空間に取り込んだ後の実行モデルとしては，3.2.1項と同様の選択肢がある．本論文では，新機能の実行時間は，メッセージ通信時間と同程度かあるいは短いことを想定していることと，カーネルタスクとの間のメッセージ通信のオーバヘッドを考えて，3.2.1項で述べた方法(b)を採用する．

(3) 新機能の追い出し

上記(2)でカーネル空間に取り込まれた新機能をユーザ空間に追い出す．追い出しはシステムコールとして提供する．

(i) 追い出し時期

追い出し時期は，当該追い出し対象の新機能が使用中でないときであるが，上述したように，あるユーザタスクが追い出しシステムコールを発行したときは，カーネル自体がアイドル状態であるので，当該新機能が使用中か否かを調べる必要はなく，追い出しシステムコールが発行された時点で追い出してよい．

(ii) 追い出し後の実行モデル

追い出し後は，1つのユーザタスクとして動作するモデルを採用する．このとき，コンテクストは取り込み直前のコンテクストを引き継ぐモデルを採用する．

(4) 置換

(i) 割当て資源の引き継ぎ

置換元のユーザタスクを置換先のユーザタスクに置き換えるとき，置換元のユーザタスクに割り当てられているカーネルの資源（ファイル，デバイス，シグナルなど）を，置換先のユーザタスクに引き継がせるべきかが問題となる．引き継がせるとした場合は，引き継がせるカーネルの資源の種類とその実現機構は，慎重に注意深く決定しなければならない難しい問題である．本論文は，動的保護をも考慮したOSサーバの動的構築機構を統合的に行うことを目的としているので，割当て資源の引き継ぎは行わないことにする．引き継ぎの問題に関しては，今後の課題とする．

(ii) 置換時期

置換時期は，置換元ユーザタスクが使用されていないときである．ユーザタスクは実行中にカーネルスケジューラによりタスク切替えが生じる可能性があるので，上記の新機能の取り込み時期で述べたフラグにより，使用中でないことを判断して行う必要がある．

(5) 新機能の実行

追加した新機能はユーザに提供される．その新機能

表1 システムコールの形式
Table 1 Format of system calls for dynamic reconfiguration operations.

システムコールの種類	渡す引数	返り値
登録情報獲得	情報格納先の先頭アドレス (*1)	成功：0，失敗：−1
登録	構造体のアドレス (*2)	成功：0，失敗：−1
取り込み	取り込みたい機能の名前 (*3)，対象タスクを区別するフラグ (*4)	成功：0，失敗：−1
追い出し	追い出したい機能の名前 (*3)，対象タスクを区別するフラグ (*4)	成功：0，失敗：−1
新機能置換	置換したい機能の名前 (*3)	成功：旧タスクの pid，失敗：−1
カーネルスケジューラ置換	置換先のスケジューラの先頭アドレス	成功：旧タスク pid か 0 (*6)，失敗：−1
新機能（タスク）呼び出し	呼び出したいタスク名，関数名，新機能の関数が利用する引数 (*5)	成功：0，失敗：−1

(*1)：情報格納先は構造体になっており（カーネルタスクもユーザタスクも同じ構造体の変数を持つ），その先頭アドレスを渡す．
(*2)：登録する情報（タスク名，新機能の関数名，その先頭アドレスなど）は，構造体 ProcedureData の中に格納されているので，その先頭アドレスを渡している．
(*3)：機能の名前は登録システムコールにより，登録した名前であり，対象が自身の場合は，名前は “(NULL)” となる．
(*4)：対象が自分の場合は 0，他タスクの場合は 1 となる．
(*5)：引数は，実際には内容を渡すのではなく，一連の引数の先頭アドレスを格納している構造体の先頭アドレスを渡す．
(*6)：0 は，自身が初めての置換のとき，それ以外は，すでにカーネルスケジューラとして登録されているタスクがあるので，その pid を返す．

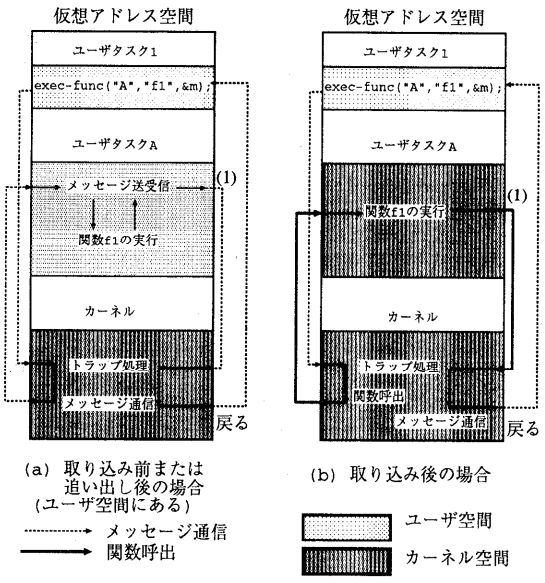


図4 新機能の実行の流れ
Fig. 4 Flow of executing a new task.

の実行の流れを，図4の例で説明する．図4は，新機能である関数名が f1 で，ユーザタスク A にあるときに，新機能をカーネル空間に取り込む前または追い出した後の場合（図4(a)），取り込み後の場合（図4(b)）について示している．ユーザは本機能を exec-func システムコール（後述する表1の新機能呼び出しシステムコールである）を用いて利用できる．図4中の (1) の通信は，(a) ではメッセージ通信，(b) では関数呼び出しになっていることに留意されたい．

3.4 カーネル内スケジューラの動的置換

スケジューラの動的置換もユーザタスクからのシステムコールの形で行う．

置換の時期については，前記と同様に，カーネルスケジューラが使用中でないときであるが，サーバタスクの場合と同様に，システムコール発行時には，カーネルはアイドル状態なので，スケジューラが使用中か否かのフラグは必要ない．

4. 実現方法

本章では，上記の実現方法について述べる．

4.1 新たに設けるシステムコール

新たに設けるシステムコールを以下に示し，その形式を表1に示す．

(1) 登録情報獲得：下記の登録システムコールにより，登録されている情報（タスク名，関数名など）をユーザタスクが獲得するためのシステムコールである．本システムコールは，置換するために置換元タスクの情報をユーザがあらかじめ知っておく場合に用いられる．

(2) 登録：本システムコールは，以下の2つの場合に利用する．

(i) 新機能をカーネルが提供するサービスとしてユーザに提供させたい場合である．すなわち，新機能をカーネルサービスとして追加する場合である．このとき，登録された新機能を含んだユーザタスクは，当該機能を提供する新たなサーバタスクとして動作することになる．

(ii) 後で述べる置換のために，置換先タスクをカー

ネルに認識させておきたい場合である．このときは，カーネルがサービスを依頼するのは，置換先タスクではなく，まだ置換元ユーザタスクに対してである．

上記いずれの場合も，本システムコールで指定されたユーザタスクは，カーネル空間への取り込みは行わず，まだユーザ空間で動作させたままである．なお，2章のモデルにおけるサーバタスクについては，本OSサーバの立ち上げのときに登録するものとし，動的な登録は行わない．登録のみを行ったタスクに対しては，カーネルはメッセージ通信で処理を依頼する．

(3) 取り込み：サーバタスクおよび登録した新機能をカーネル空間に取り込むためのシステムコールである．取り込みにあたっては，まず登録システムコールにより登録しておく必要がある．

(4) 追い出し：前記 (3) で取り込んだものをユーザ空間に追い出すためのシステムコールである．

(5) 新機能置換：登録システムコールによりユーザ空間で動作しているユーザタスク（置換元タスク）を新たなユーザタスク（置換先タスク）と交換するためのシステムコールである．(2) で述べたように，置換先タスクは，置換に先だって，登録システムコールを用いてカーネルに認識させておく必要がある．

(6) カーネルスケジューラ置換：カーネルスケジューラを動的に置換するためのシステムコールである．

(7) 新機能呼び出し：登録/追加システムコールの引数で指定されたユーザタスク内の関数を，カーネルのサービスとして利用するためのシステムコールである．ユーザには，登録/追加された関数が新しいシステムコールとしてみえる．図4中に示したexec-funcである．

4.2 必要なデータ構造

実現に際して以下のデータ構造が必要となる．

(1) ユーザタスク内データ構造

新機能の追加に用いるために，データ構造体ProcedureDataをユーザタスクに用意する．この中には，登録したいユーザタスク名，当該ユーザタスク内の追加したい新機能（関数）名，その先頭アドレス，などを格納する．

(2) カーネル内データ構造

カーネル内のユーザ/サーバタスク対応に設けるデータ構造を**図5**に示す．図5中で，データ構造ReconfigData，ProcedureDataが，動的構築のためのデータ構造である．データ構造ProcedureDataはユーザタスク内のそれと同じであり，ReconfigDataは以下のデータからなる．

・reg-flag：当該タスクが登録されているかどうかを示すフラグである．フラグのセットは，後述する登録システムコールにより行う．

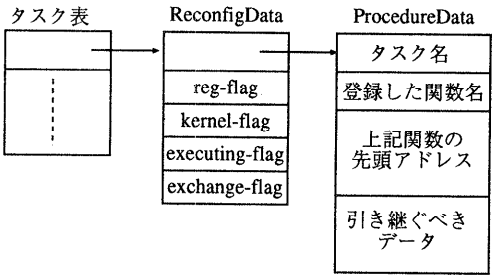


図5 タスク対応のデータ構造
Fig. 5 Data structure associated with task.

・kernel-flag：当該タスクがカーネル空間に取り込まれているか否かを示すフラグである．取り込みシステムコールによりセットされる．

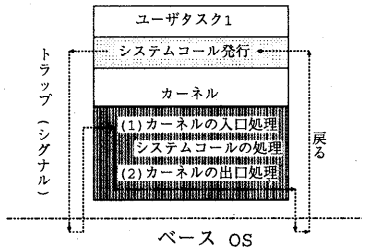
・executing-flag：当該タスクが使用中か否かを示すフラグであり，新関数の取り込み/置換の際に用いられる．

・exchange-flag：置換に絡んだ処理に使うフラグである．カーネルがサービスを依頼する相手は，置換システムコールが発行されていない場合は，置換元タスク，発行以後は，置換先タスクであるので，新機能呼び出しシステムコールをカーネルが処理する場合に，カーネルがどちらのタスクに処理を依頼すべきかを決めるときに用いられる．

なお，上記のkernel-flagは，サーバタスク内にも保持しており，カーネルタスクとの通信方法の選択に用いる（サーバタスクが取り込まれていない場合はメッセージ通信，取り込まれている場合は関数呼び出し，のどちらのコードを実行するかの判断に用いる）．

4.3 動的保護からみたシステムコールの流れ

ユーザタスクから発行されるすべてのシステムコールは，カーネルで処理（場合によってはサーバタスクにも依頼）されることになるが，ベースOSへのトラップ（UNIX系では，シグナル）を介して，カーネルに入り，システムコールを処理した後，カーネルから出ることになる．このとき，カーネル空間に取り込まれたユーザタスク，サーバタスクの動的保護の観点からの処理の流れを**図6**に示す．図6に示したように，カーネルの入り口処理では，取り込みシステムコールによりカーネル空間に取り込まれているユーザタスク，およびサーバタスクを個別にベースOSが提供している保護システムコール（UNIX系ではmprotectシステムコール）を用いて，取り込まれているタスクの領域を読み書き実行可能とする．カーネルから出るとき




```
(1)・ユーザタスク1の領域をアクセス可にする(*)(A);
   ・カーネルの領域を読み書き実行可能にする(*)(A);
   ・for(全てのユーザ/サーバタスクについて){
      if(当該ユーザ/サーバタスクがカーネル空間に
                取り込まれている)
        then 当該タスクの領域を読み書き実行可能にする(*);
        else 何もしない;
   }
(2)・ユーザタスク1の領域を読み書き実行可能にする(*)(A);
   ・for(全てのユーザ/サーバタスクについて){
      if(当該ユーザ/サーバタスクがカーネル空間に
                取り込まれている)
        then 当該タスクの領域をアクセス不可にする(*);
        else 何もしない;
      }
   ・カーネルの領域をアクセス不可にする(*)(A);

(*):アクセス不可，読み書き実行可能は，ベースOSへ
    mprotect()システムコールを発行することによって行なう.
```

図6 動的保護から見たシステムコールの処理
Fig. 6 Processing of a system call from dynamic protection view.

（カーネルの出口）では，当該タスクをアクセス不可とする．これにより，動的保護を実現する．なお，2章で仮定したOSサーバが，図6で行っている処理は(A)の処理のみであり，(A)以外の処理は本論文による動的保護機構である．むろん，その他の箇所にも動的保護機構を有している．

4.4 安全性

安全性については，登録の際に，3.1節(2)で述べたようなチェックを行う．すなわち，登録システムコールを発行したユーザタスクがルート権限で走っているかどうかをカーネルがチェックし，ルート権限であれば登録を認め，そうでなければ登録を拒否する．

4.5 登録システムコール

登録システムコールを発行する前には，以下の手続きを行っておく必要がある．

・追加すべき新機能（関数）を含んだユーザタスクにおいて，登録すべきタスク名，関数名および当該関数の先頭アドレス，および後で置換のために引き継がせたいデータがあればそれも，当該ユーザタスク内のデータ構造体ProcedureDataに格納する．

登録システムコールの処理概要を**図7**に示す．登録システムコールを発行したタスクがルート権限で走っているかチェックし，そうであれば，さらに，追加のための登録か，置換する目的での登録かのチェックを行い，各々に対応した処理を行う．

4.6 サーバタスクおよびユーザタスクの取り込みシステムコールと追い出しシステムコール

(1) 取り込みシステムコール

サーバタスクまたは新機能を含んだユーザタスクをカーネル空間へ取り込むための取り込みシステムコールの処理概要を**図8**に示す．カーネル内のkernel-flag（取り込み対象がサーバタスクであれば，サーバタスク内のkernel-flagも）をオンにする．また，取り込み対象のタスクがユーザタスクであれば，当該ユーザタスクが使用中か否かのチェックも行う．また，動的保護に関しては，mprotectシステムコールを用いて，当該タスクの領域を読み書き実行可能にする．

(2) 追い出しシステムコール

上記で取り込んだサーバタスクまたはユーザタスクをユーザ空間に追い出すための追い出しシステムコールの処理概要を**図9**に示す．基本的には，取り込み処理とは逆の処理であるが，取り込み処理のような，対象となるユーザタスクが使用中か否かのチェックは必要ない．

4.7 新機能置換システムコールとカーネルスケジューラ置換システムコール

登録システムコールにより追加されてユーザ空間で動作している新機能を，新たな新機能に置換するための新機能置換システムコールの処理概要を**図10**，およびカーネルスケジューラ置換システムコールの処理概要を**図11**に示す．

なお，カーネルスケジューラの動的置換については，複数のユーザタスクが当該置換システムコールを発行すると，自分の指定したスケジューリング方式に現在なっているとは限らず，一貫性の問題が生じる．本論文では，スケジューラ動的置換については，その基本的な実現とその効果を目的としているため，特に一貫性の対処はしておらず，最新のスケジューラに置き換えられることになる．この問題については，今後の課題とする．

4.8 新機能呼び出しシステムコール

新機能呼び出しシステムコールの処理概要を**図12**に示す．新機能が置換されているか否かをexchange-flagにより判断し，その判断結果によって決まったタスクへメッセージ通信により処理を依頼することになる．

5. 実装

OSサーバとして，2章のモデルに該当するMINIXサーバを対象とした．まず，C言語で記述されたMINIXサーバを，モジュール化する目的でC++言語

```
if ( 登録を要求して来たタスクの実 uid が root でない ) then { 失敗 (−1) を返す; }
if ( 登録で指定されたタスク名が既にカーネルに登録されている ) /* カーネル内の ProcedureData を調べて判別する */
   then {
      既に登録されているタスク名を旧タスク名に変更する;
      旧タスク名の exchange-flag をオン;
   }
reg-flag をオン;
登録で指定された情報をカーネルに登録する;/* 当該タスク内の ProcedureData をカーネル内の ProcedureData にコピー */
カーネルから，当該タスクの ProcedureData の中に，カーネルが公開する情報をコピーする;
成功 (0) を返す;
```

図 7 登録システムコール内の処理
Fig. 7 Internal processing of the registering system call.

```
if ( 取り込み対象のタスクが既に登録されている ) then {
   if (取り込み対象タスクがユーザタスクである (サーバタスクでない) ) then {
      if ( 取り込み対象タスクが使用中である ) then { 失敗 (−1) を返す;}
   }
   当該タスクの領域を読み書き実行可能にする;
   カーネル内の当該タスク対応の kernel-flag をオン;
   if ( 取り込み対象タスクがサーバタスクである ) then { 当該サーバタスク内の kernel-flag をオン;}
   成功 (0) を返す;
} else { 失敗 (−1) を返す; }
```

図 8 取り込みシステムコール内の処理
Fig. 8 Internal Processing of the injection system call.

```
if ( 追い出し対象のタスクが既に取り込まれている ) then {
   カーネル内の当該タスク対応の kernel-flag をオフ;
   if ( 取り込み対象タスクがサーバタスクである ) then { 当該サーバタスク内の kernel-flag をオフ;}
   当該タスクの領域をアクセス不可にする;
   成功 (0) を返す;
} else { 失敗 (−1) を返す; }
```

図 9 追い出しシステムコール内の処理
Fig. 9 Internal processing of the remove system call.

```
if ( 新タスクが既に登録されている ) /* カーネル内のデータ構造 ProcedureData を調べて判別 */
   then {
    if ( 新タスクが旧タスクから引き継ぐべきデータがある ) then {
       旧タスクの領域を読み書き実行可能にする;
       引き継ぐデータを新タスクが指定したアドレスにコピー;
       旧タスクの exchange-flag をオフ;
       旧タスクの領域をアクセス不可にする;
    }
   成功 (旧タスクの pid) を返す;
} else { 失敗 (−1) を返す; }
```

図 10 新機能置換システムコール内の処理
Fig. 10 Internal processing of the exchange system call.

```
if ( 新タスクが既に登録されている ) then {
  if ( 既にカーネルスケジューラの置換が行なわれている
        (既にカーネルスケジューラとして取り込まれているタスクがある) ) then {
    新タスクの内部のスケジューラ関数のアドレスに，カーネルスケジューラ関数のアドレスを変更する;
    成功 (旧タスクの pid) を返す;
  } else {
    タスクの内部のスケジューラ関数のアドレスに，カーネルスケジューラ関数のアドレスを変更する;
    成功 (0) を返す；
  }
} else { 失敗 (−1) を返す; }
```

図 11 カーネルスケジューラ置換システムコール内の処理
Fig. 11 Internal processing of the kernel-scheduler-exchange system call.

```
for ( ユーザタスク全部が対象 ) {
  if ( 旧タスクが存在する && 旧タスクの exchange-flag がセットされている ) then {
    旧タスクへ実行を依頼; /* このとき，旧タスクが取り込まれていれば関数呼出，
                                取り込まれていなければメッセージ通信になる */
    成功 (0) を返す； /* 処理はここまで */
  }
}
for ( ユーザタスク全部が対象 ) {
  if ( 新タスクが存在する ) then {
    新タスクへ実行を依頼; /* このとき，新タスクが取り込まれていれば関数呼出，
                                取り込まれていなければメッセージ通信になる */
    成功 (0) を返す； /* 処理はここまで */
  }
}
失敗 (−1) を返す；
```

図 12 新機能呼び出しシステムコール内の処理
Fig. 12 Internal processing of the exec-func system call.

で書き直し，上記の実現方法概要により実装した．実装した環境は，SPARCstation20（Sparc プロセッサ，60 MHz），SunOS である．MINIX サーバは，サーバタスクとして，メモリマネジャタスク（以下 MM と示す），ファイルシステムタスク（以下 FS と示す）の 2 つがある．MM はメモリ管理のほかに，多くのシステムコールの受付け窓口にもなっている．カーネルタスクとしては，MM および FS とのメッセージ通信窓口となるシステムタスク，クロック管理とスケジューリングを行うクロックタスク，および周辺装置への入出力を管理するいくつかのタスクの，合計 8 つがある．また，ソフトウェア割り込みはシグナル，タイマ割り込みは SIGALARM シグナル，アクセス保護は，ベース OS への mprotect システムコールにより実現されている．また，上記の実現にあたって，新機能を取り込むカーネルタスクはシステムタスクとした．さらに，MM，FS は取り込まれた場合，システムタスクの一部のコードを取り込んでいる．

6. 評価および応用

5 章の実装に基づいて，本章では，以下の 2 つの観点から評価する．

(i) 動的保護のオーバヘッドを定量的に把握すること．

(ii) 5 章で述べたように，サーバタスクである MM，FS をカーネル空間に取り込んだ際の効果を定量的に把握すること．この評価を保護がある場合とない場合の 2 つの場合で行う．

したがって，以下の 2 つの保護について性能を評価する．

(i) 保護を行わない場合：図 1 で示した仮想アドレス

表2 保護がない場合での各既存システムコールの実行時間（μsec）
Table 2 Execution times of each existing system call without protection (μsec).

処理またはシステムコール	なし（計）		あり (MM & FS)（計）		あり (MM or FS)（計）	
	ut	st	ut	st	ut	st
メッセージ通信	257		257		257 (MM)	
	102	155	102	155	102	155
getuid	274		274		274 (MM)	
	111	163	116	158	125	149
chdir	424		423		409 (FS)	
	266	158	265	158	257	152
sbrk	565		304		308 (MM)	
	285	280	158	146	154	154
times	563		317		314 (FS)	
	276	287	153	164	174	140
open& 20 bytes read& close	1734		1200		1154 (FS)	
	1010	724	746	454	694	460

ut (=utime)：ベースOSから見たユーザ時間，st (=stime)：ベースOSのシステム時間
なし：サーバタスク (MMやFS) の取り込みなし，あり：サーバタスクの取り込みあり
(MM)：MMを取り込んだ場合，(FS)：FSを取り込んだ場合

空間全体を常時読み書き実行可能とする場合である．したがって，mprotectシステムコールはいっさい発行されない．これは，上記(i)を検討する場合に必要となる．

(ii) 保護を行う場合：本論文で述べたような動的保護を行う場合である．

さらに，スケジューラの動的置換の効果も検討する．

6.1 保護を行わない場合

(1) 基本的特性

メッセージ通信（NULLメッセージで，ユーザタスクとMMとの間の同期通信）および各システムコールの（1回あたりの）実行時間を**表2**に示す．(i) サーバタスクMM，FSの両方ともカーネル空間に取り込んでいない場合，(ii) MM，FSの両方を取り込んだ場合，(iii) MM，FSのいずれかを取り込んだ場合，の3つの場合を示す．(iii) の場合は，システムコールの実行に有利になる方（影響を与えない場合もある）を取り込んでいる．測定は，当該メッセージ通信または各システムコールを1万回発行し，ベースOSのgetrusageシステムコールを用いた．メッセージ通信，getuid，chdirは，上記(i)，(ii)，(iii)ではメッセージ通信回数が同じであるので，(i)，(ii)，(iii)による実行時間の違いはほとんどない．一方，sbrk，timesは，メッセージ通信回数は，取り込まない場合は2回であるが，取り込むことにより，sbrkではMMとシステムタスク間，timesではFSとシステムタスク間のメッセージ通信がなくなるので，メッセージ通信回数は1回と1つ少なくなるので，その分実行時間が軽減されていることが分かる．また，ファイルに関するopen&20 bytes read&closeの一連の処理では，合計2回（openで1回，readで1回）少なくなるので，大体，メッセージ通信2回分だけ，実行時間が減少していることが分かる．

サーバタスクの取り込みにより最も実行時間が短縮できた既存システムコールは，sbrkシステムコールで，取り込みなしに比べて約50%実行時間を短縮できている．

(2) 動的構築に関する性能

表3に，登録/取り込み/追い出しの各システムコールの実行時間を示す．対象を新機能を含んだユーザタスクとした場合には，ユーザタスクのデータ領域の大きさを変えた場合も示す．サーバタスクであるMM，FSの大きさは**表4**に示す．また，MM，FSの登録はMINIX立ち上げ時に行うので測定が困難であり，示していない．また，動的構築のためのシステムコールを測定する場合，厳密な測定は困難である．これは，測定する際には，当該システムコールをある程度大きな回数をループで回す必要があるが，これらのシステムコールはその意味と処理内容より，1回目と2回目以降では状況が異なるからである．たとえば，対象を新機能とした取り込みシステムコールを例にとると，2回目では，1回目ですでにMINIXのカーネル内のシステムタスクに取り込まれている新機能を再度取り込むことになる．しかし，保護がない場合では，1回目と2回目以降では，当該システムコール内で実行されるコード量には差がないので，本測定方法でもよい近似になっていると思われる．本方法を用いて，当該システムコールを1000回発行して実行時間を測定した．

表3に示すように，まず，システムコールの実行時間は，対象とするタスクの大きさの影響を受けないこ

表3 保護がない場合での動的構築のための各システムコールの実行時間（μsec）
Table 3 Execution times of each system call for reconfigurability of the operating system server without protection (μsec).

(a) 新機能を対象とした場合
（新機能のテキスト領域はいずれも 25 KB）

データ領域の大きさ	16 KB		100 KB		200 KB	
システムコール	実行時間（計）		実行時間（計）		実行時間（計）	
	ut	st	ut	st	ut	st
登録	1319		1339		1329	
	908	411	900	439	864	465
取り込み	548		551		551	
	260	288	262	289	263	288
追い出し	547		551		551	
	257	290	250	301	251	300

(b) サーバタスクを対象とした場合

対象サーバ	MM		FS	
システムコール	実行時間（計）		実行時間（計）	
	ut	st	ut	st
取り込み	554		550	
	277	277	260	290
追い出し	560		560	
	275	285	289	271

ut (=utime)：ベース OS から見たユーザ時間
st (=stime)：ベース OS のシステム時間

表4 各サーバタスクの大きさ
Table 4 Area size of each server task.

	MM	FS
テキスト領域	305 KB	261 KB
データ+スタック領域	200 KB	16 KB

とが分かる．これは，予想できるように，当該システムコール内でアクセスされるデータ領域の大きさは，タスクの大きさにほとんど影響されないからである．また，各システムコールの実行時間を比べると，取り込み，追い出しに比べて，登録が大きくなっている．これは，MM とカーネルとの間でのメッセージ通信が，取り込み/追い出しでは 1 回であるのに対し，登録では，カーネルにまず登録するプロセス id を，登録情報の通知に先立って教える必要があり，このために，取り込み/追い出しに比べて，MM ⟷ カーネル間のメッセージ通信が 1 回多くなるためである．

6.2 保護を行う場合

以下で示す実行時間の測定方法は，各々6.1 節のそれと同じである．

(1) 基本的な特性

表 5 に，保護を行った場合の各システムコール実行時間およびベース OS への mprotect システムコール発行回数を示す．また，カーネル，サーバタスクおよびユーザタスクの保護に関する操作（具体的には，mprotect の発行回数）に着眼したメッセージ通信処理の流れを，**図 13** に示す．

(i) メッセージ通信性能

保護を行った場合（表 5）のメッセージ通信処理（ユーザ空間にある MM とユーザタスクとの間での NULL メッセージ）は，保護を行わなかった場合（表 2）よりもかなりの時間がかかっている．これは，図 13 から分かるように，1 回のメッセージの送受信には，保護に用いる mprotect システムコールの発行回数が計 47 回と多く，mprotect システムコールの実行時間が大きな影響を与えているためである．また，MM，FS をカーネル空間に取り込んだ効果をみると，保護を行わない場合では，MM，FS を取り込んでも，メッセージ通信のオーバヘッドが変わらないのに比べ，保護を行った場合では，MM/FS を取り込むことによってメッセージ通信処理時間が減少している．これは，MM/FS がカーネル空間となるので，MM/FS に関する保護操作（mprotect の発行回数）が軽減されるためである（表 5 参照）．

MM と FS の両方を取り込んだ場合と，MM のみを取り込んだ場合とを比較すると，MM のみの方が若干性能がよい．これは，FS をも取り込んだ場合では，カーネル空間へ空間が切り替わったとき，FS の保護のために mprotect システムコールを発行するため，MM のみの取り込みに比べて，mprotect の発行回数が多くなるためである．mprotect の発行回数は，取り込みなしでは 47 回，MM と FS の両方を取り込んだ場合は 37 回，MM のみの取り込みの場合は 29 回となっている．

(ii) 各システムコールの性能

sbrk，times，ファイルへの読み込みでは，前述したように，MM/FS の取り込みによりメッセージ通信回数が減少するので，実行時間も減少している．getuid と chdir は，メッセージ通信回数が取り込みなしと変わらないのにもかかわらず，取り込んだ場合の実行時間が減少している．これは，6.2 節 (1)(i) で述べたように，mprotect 発行回数が少なくなるためである．これは，保護を行う場合の特徴的な現象である．

また，サーバタスクの取り込みにより最も実行時間が短縮できた既存システムコールは，保護を行わない場合でもそうであったように，sbrk システムコールで，取り込みなしに比べて約 60%に実行時間を短縮できている．

(2) 動的構築に関する性能

登録/取り込み/追い出しのシステムコールの実行時

表5 保護がある場合での各既存システムコールの実行時間（msec）とmprotectシステムコールの発行回数
Table 5 Execution times (msec) and the number of the mprotect system calls of each system call with protection.

処理またはシステムコール	なし（計）		あり (MM & FS)（計）		あり (MM or FS)（計）	
	ut	st	ut	st	ut	st
メッセージ通信	7.0 [47]		6.1 [37]		5.8 (MM)[29]	
	1.4	5.6	1.1	5.0	1.1	4.7
getuid	7.2 [47]		6.3 [37]		6.2 (MM)[29]	
	1.5	5.7	1.3	5.0	1.3	4.9
chdir	8.1 [47]		7.1 [37]		6.7 (FS)[29]	
	1.7	6.4	1.3	5.8	1.4	5.3
sbrk	11.0 [64]		6.7 [37]		6.3 (MM)[29]	
	2.2	8.8	1.3	5.4	1.3	5.0
times	10.9 [64]		6.6 [37]		6.5 (FS)[29]	
	2.3	8.6	1.1	5.5	1.2	5.3
open& 20 bytes read& close	26.6 [200]		22.0 [141]		21.3 (FS)[115]	
	5.3	21.3	4.1	17.9	4.0	17.3

ut (=utime)：ベースOSから見たユーザ時間，st (=stime)：ベースOSのシステム時間
[] 内はmprotectシステムコール発行回数，なし：サーバタスクの取り込みなし，あり：サーバタスクの取り込みあり
(MM)：MMを取り込んだ場合，(FS)：FSを取り込んだ場合

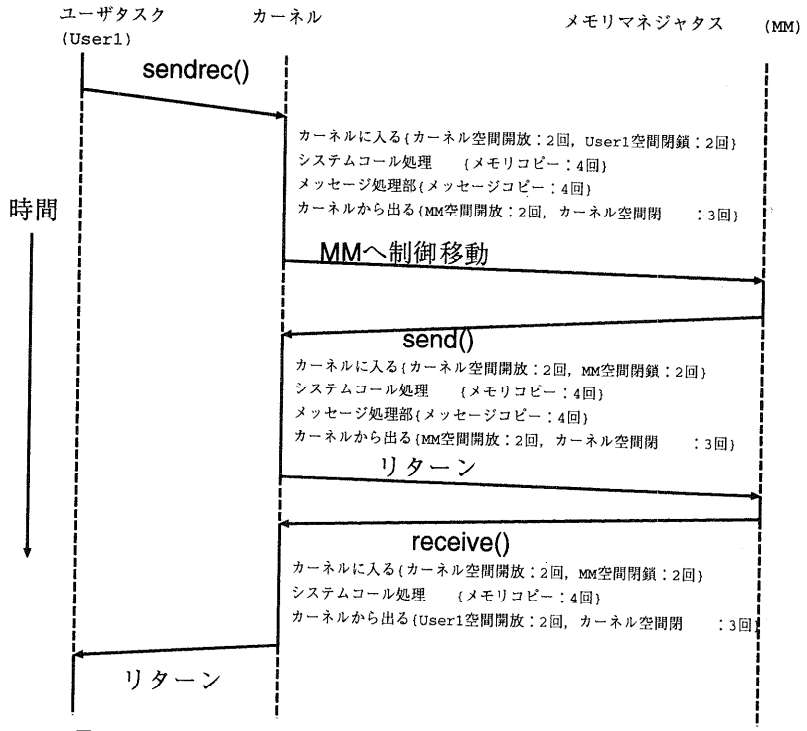


図13 通常のメッセージ通信の流れとmprotectシステムコールの発行回数
Fig. 13 Message communication flow and the number of the mprotect system calls issued.

間に関しては，保護を行わない場合は，6.1節(2)で述べたように，システムコールの1回目の発行と2回目以降との間には，処理内容に差がないが，動的な保護を行う場合には，mprotectシステムコールの発行回数に違いがでる．たとえば，取り込み対象を新機能とした場合での取り込みシステムコールの処理を例にとると，カーネル空間への入り口/出口でのmprotect発行回数が異なる．2回目以降のシステムコールの発行時点を考えると，1回目でカーネル空間にすでに取り込まれている機能を再度取り込む状況となる．したがって，2回目以降では，カーネルの入り口/出口では1回目と比較して，すでにに取り込まれている新機能に関

表6 保護がある場合での動的構築のための各システムコールの実行時間（2回目以降，msec）とmprotectシステムコール発行回数

Table 6 Execution times (msec) and the number of the mprotect system calls issued of each system call for reconfigurability of the operating system server with protection.

(a) 新機能を対象とした場合
（新機能のテキスト領域はいずれも 25 KB）

データ領域の大きさ	16 KB		100 KB		200 KB	
システムコール	実行時間（計）		実行時間（計）		実行時間（計）	
	ut	st	ut	st	ut	st
登録	14.5 (87)[87]		15.0		15.1	
	3.5	11.0	3.6	11.4	3.4	11.7
取り込み	12.1 (88)[94]		11.6		12.4	
	2.5	9.6	2.3	9.3	2.9	9.5
追い出し	11.9 (82)[76]		11.3		11.2	
	2.6	9.3	2.4	8.9	2.2	8.9

(b) サーバタスクを対象とした場合

対象サーバ	MM		FS	
システムコール	実行時間（計）		実行時間（計）	
	ut	st	ut	st
取り込み	9.2 (58)[47]		12.1 (88)[94]	
	1.8	7.4	3.0	9.1
追い出し	10.9 (63)[74]		11.0 (82)[76]	
	2.2	8.7	2.3	8.7

ut (=utime)：ベースOSから見たユーザ時間
st (=stime)：ベースOSのシステム時間
() 内はシステムコール1回目のときのmprotectシステムコール発行回数
[] 内はシステムコール2回目以降のmprotectシステムコール発行回数

しても余分に動的保護を行うため，この分mprotectシステムコールの発行回数が多くなる．したがって，この差を最終的には補正する必要がある．

まず，**表6**に，登録/取り込み/追い出しの各システムコールについて，ループの2回目以降を測定したときの実行時間とmprotect発行回数，および1回目のmprotectシステムコールの発行回数の実測値を示す．これらの数値とmprotectシステムコール実行時間（実測値は90 μsec）を用いて補正して得られた1回目発行の当該システムコールの実行時間を，**表7**に示す．

まず，保護を行わない場合（表3）と行う場合（表7）を比較すると，保護を行った場合の方が，メッセージ通信自体のオーバヘッドが大きいため，動的構築のためのシステムコールの実行時間が，保護を行わない場合よりも約20倍長くなり，かなり大きくなっている．

また，登録/取り込み/追い出しの対象を新機能とした場合のその大きさの違いによる影響をみる．mprotect発行回数は，新機能の大きさに依存しないので，

表7 動的構築のための各システムコールの実行時間（1回目の概算値，msec）

Table 7 Estimated execution times (msec) of each system call for reconfigurability of the operating system server with protection.

システムコール	新機能	MM	FS
取り込み	11.6 (88)[94]	10.2 (58)[47]	11.6 (88)[94]
追い出し	12.4 (82)[76]	9.9 (63)[74]	11.5 (82)[76]

データ+スタック領域は，新機能：16 KB，MM：200 KB，FS：16 KB
() 内は1回目のmprotectシステムコールの発行回数
[] 内は2回目以降のmprotectシステムコールの発行回数

表6(a)でその違いを考察できる．表6(a)より，各システムコールの実行時間は，対象を新機能とした場合では，その大きさにあまり影響されないことが分かる．次に，対象を新機能，MM，FSとした場合での対象の違いによる各システムコールの実行時間を比較する．表7より，対象をMMとした場合の方が，新機能およびFSの場合よりも各システムコールの実行時間が短くなっている．これは，本実装法ではMMをシステムコールの受け口にしていることに起因して，MMの場合の方が，mprotect発行回数が少なくなるためである．保護を行わない場合では，各システムコールの実行時間は，対象とするタスクの種類（新機能，サーバタスク）およびその大きさの影響を受けなかったのに対し，動的保護を行うと，ここでは，MMの場合が短くなるという，タスクの種類による影響を受けることを示唆している．

6.3 スケジューラの動的置換

カーネルスケジューラの動的置換の効果を評価する．ここでは，1つの応用プログラム内でのスレッド群の実行パターンが異なる，すなわち，適したスケジューリング方式を動的に変更した方がよい応用プログラムについて，スケジューラの動的置換の効果を調べる．適用する応用プログラムの実行パターンを図**14**に示す．MINIXのforkシステムコールでは，子スレッドが親スレッドよりも先にスケジューリングされる．フェーズ1ではFIFO方式が適している．フェーズ2は，スレッド(E)が(C)または(D)のどちらか一方の実行終了を待つパターンであり，(C)と(D)のRR方式が適している．スレッド(A)および(D)の実行時間はどちらとも大体12秒である．図**15**に，すべてFIFO方式，すべてRR方式，フェーズ1にFIFO，フェーズ2にRRを採用した場合の（フェーズ1と2の間で動的置換を行う），応用プログラムの実行時間（フェーズ1とフェーズ2の時間の和）を示す．スレッド(C)の実行時間を振らせており，横軸は，スレッド(D)の

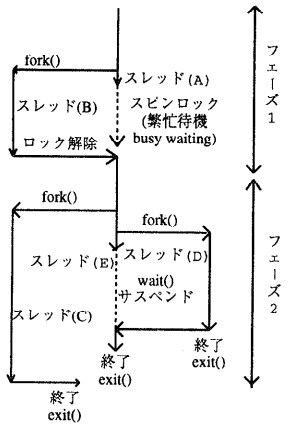


図 14 応用プログラム例
Fig. 14 An application program.

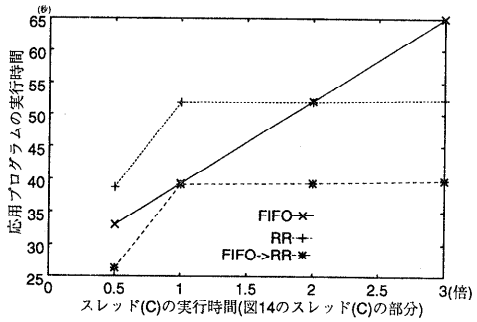


図 15 各スケジューラを用いた場合の応用プログラムの実行時間
Fig. 15 Execution times of an application program with each scheduling policy.

実行時間に対するスレッド (C) の実行時間の倍率を示している．予想されるようにFIFOからRRへの動的置換により，応用プログラムの実行時間が最も短くなることを，図15は実証している．

また，図14では，ロックの取り方として，スピンロックを用いているが，この意義について考察する．単一プロセッサの計算機上の応用プログラム内では，スピンロックではなく，サスペンドロックを用いる場合が多いと考えられる．これは，通常はスケジューリング方式が固定なので，スピンロックによる繁忙待機（busy waiting）する待ち時間が見積もれずかつ長くなる可能性があり，プロセッサを浪費するためである．しかし，もしも，動的にスケジューリング方式を変えることにより，繁忙待機時間がなくなれば，繁忙待機しない代わりにサスペンドさせるオーバヘッドがあるサスペンドロックよりも，スピンロックを用いる方が有利となる．本例題は，その有効性を示唆したものである．

6.4 応用に関する議論

本節では，本論文で述べた機構の応用について若干議論する．

(1) 新機能の追加/取り込み/置換について

追加する新機能が，カーネルデータへの読み込みアクセスのみを行う（書き込みアクセス不可）処理であれば，新機能の動的追加/取り込みは安全である．この制限は厳しいかもしれないが，この制限下でのいくつかの応用が考えられる．たとえば，ユーザタスクのスケジューリング情報（サスペンドされているか実行可能状態か，さらに，実行可能状態であればレディキューでの位置など）を通知してもらう新機能（カーネルデータへは読み込みのみ）を動的に追加して，この情報をユーザタスクが同期に活用するなどの応用が考えられる．

(2) スケジューラの動的置換について

本論文では，FIFOからRRへの動的置換の実行例しか示していないが，その他への置換例として，RRにおけるタイムスライスをユーザタスクごとに異なるようにするスケジューリング方式への動的置換も実現できる．ただし，このとき，タイムスライスの長さは，カーネルが定義しているタイムスライスの整数倍という制限がある．

7. おわりに

本論文では，動的保護が可能な動的構築機構を有するOSサーバを実現し，性能評価を行った．動的構築機構としては，動的保護をも考慮して，サーバタスクのカーネル空間への取り込み/追い出し，新機能の動的追加（登録）/取り込み/追い出し/置換，カーネルスケジューラの動的置換，である．これらを実現するための選択肢・検討課題を整理するとともに考察を加え，本論文で採用するアプローチを示した．さらに，これに基づいて，実際に既存のOSサーバを修正する方向で，動的構築機構を実装した．さらに，性能評価を行い，以下を示した．

(i) サーバタスクのカーネル空間への取り込みにより，メッセージ通信回数および動的保護処理が軽減され，システムコールの実行時間を減少できる．我々の実装では，最大約半分に実行時間を短縮できることを示した．

(ii) 動的保護処理のオーバヘッドはかなり大きい．我々の実装では，保護を行わない場合に比べて，約20倍のオーバヘッドがある．

(iii) カーネルスケジューラの動的置換は効果がある．

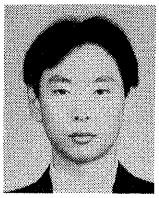
今後の検討課題としては，動的保護処理のオーバ

ヘッドの軽減，動的構築におけるカーネルの安全性の検討などがある．

参考文献

1) Bershad, B.N., et al.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symp. on Operating Systems Principles*, pp.267–284 (1995).
2) Moriai, S. and Tokuda, H.: Dynamic Loadable Object Support for Real-Time Mach Kernels, *Proc. World Computing and Its Applications (WWCA '97)*, pp.318–333 (1997).
3) Mahmoud, M. and El-Kadi, A.: A DOS/Linux Extensible File System, *Proc. 2nd Symp. on Computers and Communications*, pp.311–315 (1997).
4) Burns, J., et al.: A Dynamic Reconfiguration Run-Time System, *Proc. 5th Annual Symp. on Field-Programmable Custom Computing Machines*, pp.66–75 (1997).
5) Seltzer, M. and Small, C.: Self-monitoring and Self-adapting Operating Systems, *Proc. 6th Workshop on Hot Topics in Operating Systems*, pp.124–129 (1997).
6) Clark, M. and Coulson, G.: An Architecture for Dynamically Extensible Operating Systems, *Proc. 4th Int'l Conf. on Configurable Distributed Systems*, pp.145–155 (1998).
7) 多田好克，中村嘉志，林　隆宏：カーネルの発展性と安全に関する一考察，情報処理学会 OS 研究会，97-OS-76, pp.61–65 (1997).
8) 谷口秀夫，伊藤健一，牛島和夫：プロセス走行時におけるプログラムの部分入替え法，信学論 (D-I)，Vol.J78-D-I, No.5, pp.492–499 (1995).

(平成 10 年 12 月 4 日受付)
(平成 11 年 4 月 1 日採録)



柏木　一彦（学生会員）

1969 年生．1994 年立命館大学理工学部情報工学科卒業．1996 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．現在同大学博士後期課程在学中．オペレーティング・システムに関する研究に従事．

福田　晃（正会員）

1954 年生．1977 年九州大学工学部情報工学科卒業．1979 年同大学院工学研究科修士課程修了．同年 NTT 研究所入所．1983 年九州大学大学院総合理工学研究科助手．1989 年同大学助教授．1994 年より奈良先端科学技術大学院大学情報科学研究科教授，工学博士．オペレーティング・システム，並列化コンパイラ，計算機アーキテクチャ，並列/分散処理，性能評価等の研究に従事．本学会平成 2 年度研究賞，平成 5 年度 Best Author 賞受賞．著書「並列オペレーティングシステム」(コロナ社)，訳書「オペレーティングシステムの概念」(共訳，培風館)．AMC，IEEE Computer Society，電子情報通信学会，日本 OR 学会各会員．

最所　圭三（正会員）

1959 年生．1982 年九州大学工学部情報工学科卒業．1984 年同大学院工学研究科修士課程修了．同年同大学工学部助手．1991 年同大学工学部講師．1993 年同大学大型計算機センター助教授．1994 年奈良先端科学技術大学院大学情報科学研究科助教授，現在に至る．工学博士．高信頼性システム，並列/分散処理，モバイルシステム，並行処理等の研究に従事．1998 年情報処理学会全国大会大会優秀賞受賞．電子情報通信学会会員．