システムソフトウェアと
オペレーティング・システム 77－35
マルチメディア通信と分散処理 87－35
（1998. 2. 27）

カーネルスケジューラの動的置換

柏木 一彦　田中 義照　最所 圭三　福田 晃
奈良先端科学技術大学院大学 情報科学研究科
〒630-01 奈良県生駒市高山町 8916-5
{kazuhi-k, yosite-t, sai, fukuda}@is.aist-nara.ac.jp

あらまし
計算機の性能向上、様々な入出力装置の出現、アプリケーションの拡大などの、計算機を取り巻く環境の変化は非常に大きい。そのため、これらの変化に追従できるオペレーティングシステムが必要である。我々は、機能の一部を、動作させながら動的に組込める、削除できる、同じような機能と置換できる、オペレーティングシステムの実現を目指している。今回は、非常に制限はされているが、カーネル内の関数を動的に置き換えることができるオペレーティングシステムのプロトタイプを実装、カーネルスケジューラ内の関数の置換によるスケジューリングポリシの動的変更、および、実験による動的変更の有効性の確認を行ったので報告する。

キーワード　カーネル機能、動的変更、スケジューラ、スケジューリングポリシ、実験的評価

Dynamic Replacement of Kernel Scheduler

Kazuhiko KASHIWAGI, Yoshiteru TANAKA, Keizo SAISHO and Akira FUKUDA
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-01
{kazuhi-k, yosite-t, sai, fukuda}@is.aist-nara.ac.jp

Abstract
By drastic changing of computer environment such as improving performance of computers, developing various I/O devices, and expanding computer applications, operating systems are requested to cope with such changing. Thus, this study aims at implementing an operating system which is able to add functions to it, delete them from it, and replace them with new functions without stopping. We implement the prototype of the operating system that can replace functions in it dynamically with much limitation, show that it is possible to change the scheduling policy by replacing some function in the kernel scheduler, and confirm the availability of the replacement.

Keywords　Kernel function, Dynamic replacement, Scheduler, Scheduling policy, Experimental evaluation

1 はじめに

計算機を取り巻く環境の変化に追従するためにオペレーティングシステム (以下 OS と記す) にも発展的な機構が必要になる。新たなデバイスの出現と共に計算機の応用範囲も広がり、OS もそれらに対して敏速に適応しなければならない。そのため、我々はその構成を動的に構築可能な OS の研究を行っている [1, 2]。そこでは、OS が持つべき機能を、OS を動作させながら動的に組み込んだり、削除したり、同じような機能と置き換えたりすることを目指している。情報収集、収集した情報の処理、処理結果の出力を連続して行うような応用を考えた場合、情報収集時には実時間処理を行えるようなスケジューラが必要となり、処理中は計算機の使用効率が最大になるスケジューラが望まれ、結果の出力時にはグラフィックインターフェースが要求される。我々が提案している OS を実現できれば、情報収集からその処理に変わるときに実時間性を重視したスケジューラから利用効率を重視したスケジューラに置き換えることができる。また、出力時には必要とされるグラフィック機能を持つモジュールを OS 内に組み込むことができる。

OS に必要とされる機能 (ここでは、カーネル機能と呼ぶ) を動的に変更に関する研究もいくつかある [11, 12, 13, 14]。しかし、これらの研究は、カーネル機能を実現するシステムサーバの動的な追加や削除にのみ関している。これに対して、我々の研究では、OS の運用中に、カーネル機能をカーネル空間に取り込み、さらにカーネル機能を呼び出しの方法をメッセージ通信から関数呼出しにすることにより、呼び出しのオーバヘッドも削減している。これにより、カーネル機能の開発中には 1 つのプロセスとしてユーザ空間で動作させることによりテストやデバッグを容易に行うことができ、運用中はそのまま追加することで高速に運用できる、などの特徴を持たせることができる。

我々は、既に文献 [1, 2] において、カーネル機能を交換できることを示し、呼び出しの方法がメッセージ通信から関数呼出しに変更したことにより呼び出しのオーバヘッドを削減できることを実験的に確認した。しかし、そこでは、同じ機能を交換したので、カーネル機能の変更による効果を示していなかった。そこで、本稿では、最も重要なカーネル機能の一つであるカーネルスケジューラの中でスケジューリングポリシーに関する部分を、プログラムを動作中に置き換えることにより、処理効率を向上できることを示す。

以下、第 2節でカーネル機能の動的変更について述べ、第 3節で動的変更を行うための基本操作について述べる。さらに、動的変更の実現例として、第 4節でスケジューラの動的変更について述べる。

2 カーネル機能の動的変更

従来の OS の構成法の代表的なものとして、OS の全ての機能を 1 つのモジュールに包含する単層カーネル構成と、OS の各機能をモジュールとして構成したマイクロカーネル構成がある。前者は、カーネル機能間の呼び出しを関数呼び出しで行えるので高速であるが、1 つの機能の変更で OS 全体を変更しなければならないので、柔軟性・拡張性・移植性が低い。これらを向上させるために、後者の構成が研究されるようになった [3, 4, 5, 6, 7, 8, 9, 10]。マイクロカーネル構成の OS では、必要最小限度の機能をマイクロカーネルとして実現し、その他のカーネル機能はマイクロカーネル上で動作するシステムサーバとして提供する。ハードウェア構成の差異をマイクロカーネルで吸収することにより、移植性を向上できる。また、システムサーバをカーネル機能ごとに用意することにより 1 つの機能の変更を 1 つのシステムサーバの変更に押さえることができるので、柔軟性・拡張性が向上する。しかし、マイクロカーネルとシステムサーバ間の呼び出しをプロセス間通信を用いて行うので OS のオーバヘッドが大きくなる。このため、時間的制約の大きなアプリケーションに用いることができない場合も生じる。

我々は、OS の運用中にカーネル機能の動的な変更を行うことができる OS の実現を目指している。同様の研究として、SPIN[11]、Apertos[12]、Mach 3.0 におけるカーネル内のサーバ [13]、VINO[14] などがある。また、デバイスドライバに関してはデバイスドライバのダイナミックローディングなどがある。本研究では、デバイスドライバも含めたあらゆるカーネル機能を OS の運用中に動的に変更可能とすることを目指している。

カーネル機能を動的に変更することにより以下の

利点を生じる。

- アプリケーションの変化に追従した最適なサービスの提供

  アプリケーションが異なれば同じ種類のカーネル機能でも、要求される機能が異なる。例えば、スケジューリングを行う場合、計算のみを行い、時間的な制約がないアプリケーションに対しては、プロセッサの利用効率が最も向上するFIFOを用いたスケジューラが適している。しかし、複数のアプリケーションに対して会話的な処理を行う場合、均等にサービスできるラウンドロビンを用いたスケジューラが適している。会話的な処理が多い昼間はラウンドロビンを用いたスケジューラを用い、夜間はFIFO用いたスケジューラと置き換えることにより、計算機の利用効率を向上できる。

- メモリの利用効率の向上

  異なるアプリケーションを1つのOS上で動作させる場合、そのOSは、全てのアプリケーションが必要とするカーネル機能を提供しなければならない。そのため、OS自体が占めるメモリ空間が非常に大きくなる。そのため、組み込みシステムのように制限がある場合は、アプリケーションが使用できるメモリを圧迫する。組み込みシステムを考えた場合、搭載されているアプリケーションは多くても、同時に動作させる個数はそれほど大きくないと考えられる。このようなシステムに、本研究で提案しているOSを搭載することにより、実行中のアプリケーションに必要な機能のみを含むようにでき、OSが占めるメモリ空間を押さえることができる。

- 拡張性・柔軟性の向上

  新たなカーネル機能が必要なアプリケーションが登場した場合、その機能をOSの運用中に追加できるので、他のアプリケーションに影響しないで対応できる。また、新たなカーネル機能を開発する場合、ユーザ空間でシステムサーバとして動作させることにより、一般のアプリケーションで使用しているテスト・デバッグ環境を用いることができる。また、ユーザ空間でプロセスとして動作させているので、カーネル空間を直接壊すことができないので安全にテスト・デバッグが可能となる。

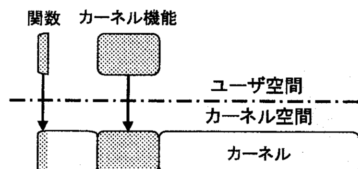

図 1: カーネル機能の追加

3 動的変更の基本操作

以下の3つの基本機能によりカーネル機能の動的変更を実現できる。カーネル機能内の一部の関数から独立したカーネル機能 (システムサーバなど) までを操作の対象としている。

(1) 追加

ユーザ空間に置かれているカーネル機能または関数をカーネル空間に移動させるものである (図1)。

カーネル機能を追加する場合は、カーネル機能を呼び出し方をプロセス間通信から関数呼び出しに変更する。このため、カーネルおよびカーネル機能は、プロセス間通信と関数呼び出しの両方のインタフェースを持ち、ユーザ空間に置かれている場合とカーネル空間に置かれている場合とで使い分けなければならない。

関数を追加する場合は、関数呼び出しを前提としているので、追加の対象となるカーネル機能に対して追加する関数の呼び出しルーチンを加える。

両方の共通の問題として、移動の対象をカーネル空間のどのアドレスにどのようにして配置するかがある。また、カーネルおよびカーネル空間に置かれている他のカーネル機能のデータ構造などをどのようにして認識させるかも問題になる。これは、メモリ空間の管理の方法に依存しており、個々に対処しなければならない。

(2) 削除

カーネル空間に置かれているカーネル機能または関数をカーネル空間から追い出す。このとき、OSから完全に削除するか (図2の①)、ユーザ空間に移動

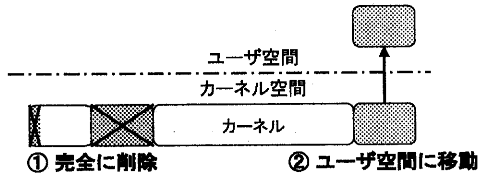

図 2: カーネル機能の削除

するだけか (図2の②) によって処理が異なる。前者は、対象のカーネル機能をそれ以降用いないことが確定している場合に生じる。この場合、対象のカーネル機能が保有していた資源を解放するので、再利用できる。後者は、カーネル機能をバージョンアップさせるときなどに、バックアップ用としてメモリ内に残すときに生じる。関数が対象になっている場合は、ユーザ空間に移動させても役に立たないので、完全に削除することになる。

(3) 置換

以下の理由により削除と追加を同時に行う。

- スケジューラやメモリ管理機構など常に動作させなければならない機能を置換する場合、削除と追加を独立して行うと空白の期間が生じOSが正常に動作しない危険性を伴う。
- 削除、追加を独立して行う場合、カーネル機能内に保持しているデータ構造を受け継ぐことができない。例えば、スケジューラを置換する場合、プロセステーブル等のデータ構造を受け継ぐ必要がある。

削除の際に古いカーネル機能を削除するか (図3の①)、ユーザ空間に移動するだけか (図3の②) によって場合分けされる。関数が対象の場合は、古い関数が新しい関数に置き換わるだけである。

4 スケジューラの動的置換

カーネルレベルの置換に関しては、文献 [1, 2] で報告している。本節では、置換の手順を示す。そのあと、スケジューラ内でランキューを操作する関数を置換することにより、スケジューリングポリシーを変更できることを示し、その効果を実験により評価する。

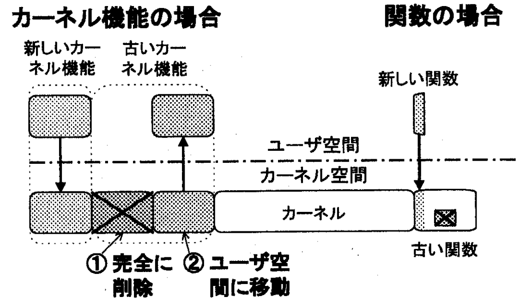

図 3: カーネル機能の置換

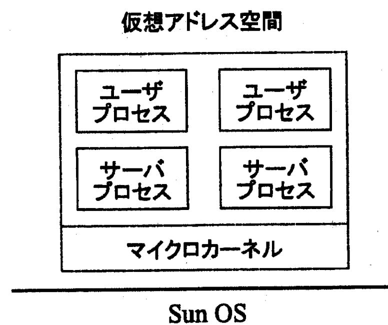

図 4: 対象の環境

4.1 関数の置換の方法

3節の追加の項目のところで示したように、置換の方法は対象の環境によって変化する。本節では、図4に示すように、対象とする環境はSun OS上の1つのプロセスとして与えられる。これにより、マシンに依存する入出力の部分をSun OSに依頼できるので、非常にシンプルになり、開発が容易になる。

カーネル、サーバプロセス、ユーザプロセスはスレッドとして実装され、1つのメモリ空間を共有する。もちろん、それらの間はSun OSのメモリ保護機能を用いて直接アクセスできないようになっている。アクセスするためには保護モードの変更が必要になるが、その機能はカーネルだけが行えるようになっている。

置換はシステムコールとして実装する。以下に置換の手順の概要を示す。

1) ユーザプロセスは置換システムコールを発行。

2) Sun OS はこれをシグナルとして本環境に通知。

3) シグナルハンドラはシステムタスク (実際にシグナルを処理する部分) に制御を渡す。

4) システムタスク内で以下の処理を行う。

   (a) *mprotect* システムコールにより、保護モードを変更することにより、新関数のアドレス範囲がカーネルから見えるようにする

   (b) 旧関数にジャンプするようななっている部分を新関数にジャンプするように書き換える

5) システムコールからユーザプロセスに戻る。

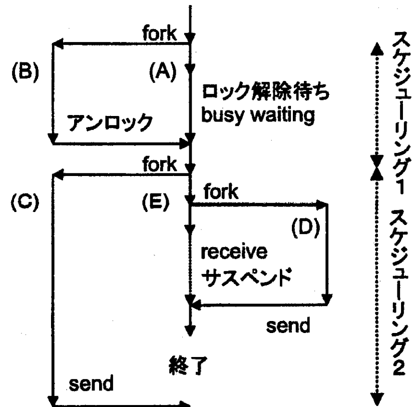

図 5: テストプログラム

4.2 スケジューラの置換

対象の環境では、ラウンドロビンによるスケジューリングが行われている。これを、FIFO でスケジューリングするように変更する。

関数の置換に際して、プロセステーブルなどのカーネル内部のデータ構造を変えることは、非常に複雑になる。今回は、カーネル内部の関数の置換が可能であること、およびその効果を調べることが目的であるので、カーネル内部のデータ構造を変えないで、スケジューリングポリシを変更する。

対象の環境では、ランキューによりどのプロセスを実行するかを決めている。現在実行中のプロセスは、ランキューの先頭に置かれている。タイマ割り込みが入ると、ランキューを操作する関数が、ランキューの先頭のプロセスを末尾に移動させることによりラウンドロビンを実現している。この関数を、何もしない関数に置きかえることにより、ランキューのデータ構造を変化させないで、スケジューリングポリシーを FIFO に変化できる。この方式は、タイマ割り込みの回数は変化しないので、そのハンドリングのオーバヘッドは変化しないが、そのオーバヘッドは非常に小さいのでほとんど問題にならない。

4.3 スケジューラの置換の効果

図 5にテストプログラムを示す。括弧で囲まれたアルファベットは、プロセスの名前を示す。最初の **fork** と 2 番目の **fork** の前に置換システムコールを呼び出して、スケジューリングポリシを変更する。FIFO でスケジューリングする場合、子供のプロセスが先に実行されるようにしている。

テストプログラムでは、前半で子プロセスに処理を依頼し、その終了を **busy waiting** によって待つ。後半では、2 つの処理を 2 つの子プロセスに依頼し、どちらか一方が終了すれば、全体の処理を終了するものである。このとき、子プロセスとの同期は、**send**、**receive** によるプロセス間通信を用いており、親プロセスはサスペンドされる。このため、前半をラウンドロビンでスケジューリングした場合、**busy waiting** での待ちに対しても、CPU 時間が割当てられるので FIFO が適している。これに対して、後半では、親プロセスはサスペンドされる、子プロセスのどちらか終了すれば親プロセスは終了のための処理を行いもう一方の子プロセスは中止される、の 2 つの理由により、ラウンドロビンが適している。それ確認するために、以下のスケジューリングポリシを変えた場合のプロセスの実効順を示す。

1) 全て FIFO

   ― (B) → (A) → (C) → (D) → (E)

2) 全てラウンドロビン

   ― (A),(B); 時分割→ (C),(D); 時分割
   → (C),(E); 時分割

3) スケジューリング 1=FIFO、
   スケジューリング 2=ラウンドロビン

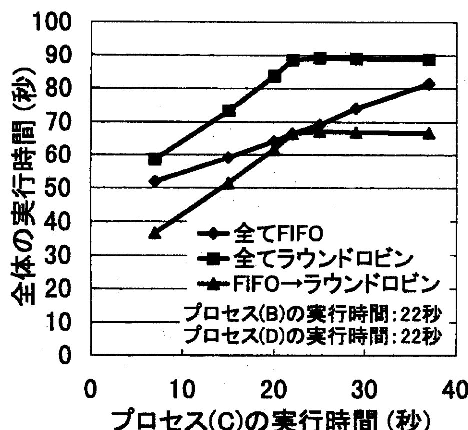

図 6: 実験結果

– (B) → (A) → (C),(D); 時分割
→ (C),(E); 時分割

このとき、プロセス (B), (D) の実行時間を固定し、プロセス (C) の実行時間を変化させて実行時間を測定した。結果を図6に示す。なお、実行順序の説明では、プロセス (C) の実行時間がプロセス (D) の実行時間より長い場合を示しているが、短い場合は、最後の時分割の組が (D),(E) に変わるだけである。後半をラウンドロビンに変更した場合の全体の実行時間は、プロセス (C) の実行時間がプロセス (D) の実行時間より短い場合は、プロセス (C) の実行時間に比例して増加するが、長くなると、一定になる。これは、プロセス (C) の処理が途中で中止されていることを示す。プロセス (C) の実行時間がプロセス (D) の実行時間が等しい場合は、両方とも完全に実効されるため、FIFO で実行した場合を同じになっている。その結果、プロセス (C) の実行時間に関係なく、スケジューリングポリシを FIFO からラウンドロビンに変えることにより実行時間が最も短くなり、スケジューラの動的変更の効果を確認できた。

5 まとめ

以上、カーネル機能の動的変更を機能内の関数を変更することにより実現する方法を示した。さらに、スケジューラ内の関数を置換し、実際にテストプログラムを実行させせることにより、その有効性を示した。しかし、今の実装は、内部に状態を持たない、旧関数を同じデータ構造を用いる、などの非常に単純な関数の置換のみ行える。今後は、このような制限を持たない実装を目指す。

参考文献

[1] 柏木一彦 他, "カーネル機能の動的変更," 情報処理学会コンピュータシステム・シンポジウム, pp.131-138, 1996.

[2] K.Kashiwagi at al, "Design and Implementation of Dynamically Reconstructing System Software," Proc. Asia Pacific Software Engineering Conference (APSEC'96), pp.278-287, 1996.

[3] D.L.Black et al., "Microkernel Operating System Architecture and Mach," Proc. USENIX Workshop on Microkernels and Other Kernel Architecture, pp.11-30, 1992.

[4] R.F.Rashid et al., "Mach: A System Software Kernel," J. Computer System Engineering, Vol.1, No.2-4, pp.163-169, 1990.

[5] D.P.Julin, "Generalized Emulation Services for Mach 3.0 – Overview, Experiences and Current Status," Proc. USENIX Mack Symposium, pp.13-26, 1991.

[6] M.Rozer et al., "Overview of the CHORUS Distributed Operating System," Proc. USENIX Workshop on Microkernels and Other Kernel Architecture, pp.39-69, 1992.

[7] D.Hildebrand, "An Architectural Overview of QNX," Proc. USENIX Workshop on Microkernels and Other Kernel Architecture, pp.113-126, 1992.

[8] J.Mitchell et al., "An Overview of the Spring System," Proc. COMPCON, pp.122-131, 1994.

[9] R.Zajcew et al, "An OSF/1 UNIX for Massive Parallel Multiprocessors," Proc. Winter USENIX, pp.449-468, 1993.

[10] 中島達夫 他, "移動計算機環境におけるアプリケーションとオペレーティング・システム支援," 情報処理, Vol.35, No.12, pp.1088-1092, 1994.

[11] B.N.Bershad et al., "SPIN – An Extensible Microkernel for Application-specific Operating System Servers," Proc. SIFOPS European Workshop, pp.74-77, 1994.

[12] Y.Yokote, "The Apertos Reflective Operating System: the Concept and its Implementation," ACM SIGPLAN Notice (USA), Vol.27, No.10, pp.414-434, 1992.

[13] J.Lepreau et al., "In-Kernel Servers on Mach 3.0: Implementation and Performance," Proc. 3rd USENIX Mach Symposium, pp.39-55, 1993.

[14] Y.Endo et al., "VINO: The 1994 Fall Harvest," Harvard Computer Center for Research in Computing Technology, Technical Report TR-34-94, 1994.

[15] A.S.Tanenbaum, "*Operating System – Design and Implementation*," Prentice Hall, 1987.