システムソフトウェアと
オペレーティング・システム 82－2
（1999．8．2）

カーネル内スケジューリングポリシの動的置換

柏木一彦[1] 藤原 真[2] 最所圭三[1] 福田晃[1]
1:奈良先端科学技術大学院大学
〒630-0101 奈良県生駒市高山町 8916 の 5
{kazuhi-k,sai,fukuda}@is.aist-nara.ac.jp
2:日立情報制御システム

計算機のハードウェアの変化に対応して，アプリケーションの使用用途が拡大しており，アプリケーションのスケジューリングに対する要求が多様化している．しかし，従来の単層カーネル構成やマイクロカーネル構成のオペレーティングシステム (以下 OS) では，そのようなスケジューリングに対する要求に迅速に対応することが困難である．そこで本研究では，従来の OS の欠点を克服するため，カーネル内のスケジューリングポリシに関する部分の関数について OS 運用中に動的な変更を可能にする OS の構成法についての考察を行なう．また，既存の OS を拡張してプロトタイプを実装し，性能評価を行ない，スケジューリングポリシの動的変更の有効性を確認した．

Dynamic Exchanging of Scheduling Policy in Kernel

Kazuhiko Kashiwagi[1], Makoto Fujiwara[2], Keizo Saisho[1], Akira Fukuda[1]
1:Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5,Takayama-cho, Ikoma, Nara, 630-0101, Japan
{kazuhi-k,sai,fukuda}@is.aist-nara.ac.jp
2:Hitachi jouhou seigyo system

Because of changing of computer hardwares, the uses of computer applications are expanding, and the demands of the scheduling of applications become diversified. However, existing operating systems are difficult to quickly meet the demands of the scheduling.

In this paper, in order to get over the weak points of existing operating systems, constructing method of operating systems, which enables to change functions dynamically about scheduling policy without stopping, is proposed.

By implementing and evaluating a prototype of the kernel, effectiveness of dynamic replacement is confirmed.

1 はじめに

昨今の計算機を取り巻く環境の変化に対応して，新たなアプリケーションの使用用途が開発されている．このような新しいアプリケーションにおいて，スケジューリングに対する要求が多様化し，今までになかった独特なものになることがある．しかし，従来のオペレーティングシステム (以下 OS) のカーネル構成では，以下のような問題が存在する．

単層カーネル構成では，スケジューリングポリシを変更する場合に，一度カーネル内の該当部分のソースコードを変更しカーネルを再コンパイルして OS を再起動しなければならず，柔軟性に欠ける．一方マイクロカーネル構成では，スケジューラをユーザ空間のシステムサーバとして運用する場合，カーネルとシステムサーバ間のプロセス間通信やアドレス空間の切替えなどのオーバヘッドが大きい．

従来のカーネル構成の欠点を克服するため，様々な研究が行なわれている [1] ～ [4]．現在，従来の OS の欠点を克服するため，カーネル機能を動的に構築することを可能とする OS を研究している [5]．本研究では，スケジューリングに着目・特化して，カーネル運用中に動的な変更を可能にすることによって，スケジューリングポリシをカーネル運用中に自由に変更することを可能とする OS の構成法についての考察を行なった [6]．さらに，そのプロトタイプ上で動作する複数のスケジューリングポリシの実装を行なった．また，プロトタイプ上において，1 つのアプリケーションを実行させ，その実行の途中でスケジューリングポリシを変更することで実行時間がどのように変化するのかについての測定を行なった．

以下，第 2 章では，従来のスケジューラに関する考察を，第 3 章では，スケジューラの動的置換に関する考察を，第 4 章では，プロトタイプを実装する上での検討事項などを，第 5 章では，実際に実装したプロトタイプに関する説明を，第 6 章では，プロトタイプの性能評価を，第 7 章では，今後の検討課題などを述べる．

2 従来の OS のカーネルスケジューラ

従来のカーネルスケジューラは，カーネル内に静的に実装されているか，マイクロカーネル構成でのシステムサーバのように，カーネル外に実装されているかである．しかし，従来の構成では以下のような欠点が存在する．

・柔軟性の欠如

カーネル内に静的に実装されている場合では，スケジューリングポリシの変更は，カーネルが提供するポリシのみに限定される．カーネル内に実装されていないポリシへの変更を行ないたい場合，カーネルの再構築をし，カーネルの運用を中断して再起動する必要がある．よって，ユーザやアプリケーションの要求に合致したスケジューリングポリシへの変更が柔軟かつ柔軟に行なえない．

・オーバヘッドの増大

システムサーバとして実装されているスケジューラにおいて，自身では実行不可能な処理は他のシステムサーバやカーネルに依頼することになる．よって他のシステムサーバやカーネルとの余分な通信のオーバヘッドやコンテキストスイッチのコストも余分に生じることになり，スケジューリング処理のコストが増大する．このコストの大きさのために，実現の困難なスケジューリングポリシが存在し得ることになる．

3 スケジューラの動的変更

前章で述べた問題点を克服するために，本研究では，カーネル内の一部の関数を OS 運用中に動的に変更できるようにすることで，スケジューリングポリシの柔軟で迅速な変更を実現することを検討する．

本章では，まずスケジューラの動的変更の概要について述べる．さらに，スケジューラの動的変更が可能な OS の設計を行なうにあたっての検討項目について述べる．

3.1 概要

カーネル内のスケジューリングポリシに依存する関数をまとめて1つのモジュールとして構成し，そのモジュールをOS運用中に別のモジュールに切替え可能とすることによって，別のスケジューリングポリシへの切替えの実現を試みる．以後，カーネルスケジューラ内のスケジューリングポリシに依存する関数のモジュールを，スケジューリングモジュール (以下モジュール) と定義する．モジュールは，カーネル実行中に動的にカーネルに登録・削除・切替えができるようにする．

以下に，モジュールの登録・削除・切替えの手順を示す．

・モジュールの登録

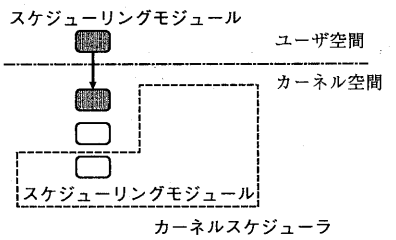

図 1: モジュールの登録

必要になったポリシを実現するモジュールをカーネルに登録する．その際，ユーザ空間内に存在する登録対象のモジュールをカーネル空間に移動させる．図1に登録のモデルを示す．

・モジュールの削除

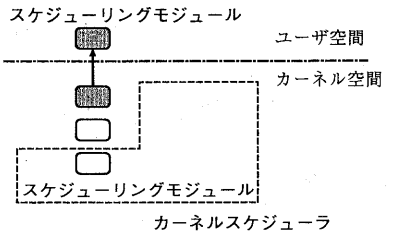

図 2: モジュールの削除

不必要になっモジュールをカーネルから削除する．その際，カーネル空間内に存在する削除対象のモジュールをユーザ空間に移動させる．図2に削除のモデルを示す．

・モジュールの切替え

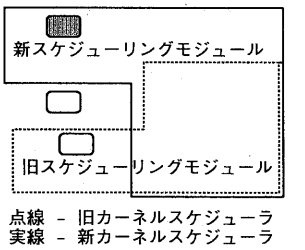

図 3: モジュールの切替え

カーネルに登録されているモジュールの中から，カーネルスケジューラが使用するモジュールを選択する．選択されたモジュール内の関数は，カーネルスケジューラ内の関数として使用される．図3に切替えのモデルを示す．

カーネルスケジューラのポリシをカーネル運用中に動的に変更できるOSには，以下のような利点がある．

・スケジューリングのオーバヘッドの軽減

本研究で提案するOSでは，モジュールの各関数は，単層カーネル構成のOSと同じく関数呼び出しで呼び出されるため，スケジューリングのオーバヘッドを低く抑えることができる．

・多様なスケジューリング要求に対する柔軟な対応

本研究で提案するOSでは，ユーザやアプリケーションの要求を受けて，カーネルが登録，削除，切替えの各操作を行なうことによりポリシを変更することが可能となる．また低いオーバヘッドでスケジューリングの処理が行なわれるので，様々なスケジューリングの要求に柔軟に対応することが可能になる．

3.2 検討項目

カーネルスケジューラのポリシを動的に変更できるOSの設計を行なうにあたって，カーネ

ル内部やモジュールに関して考慮すべき検討項目を以下に挙げる．

・モジュールの整理

モジュールを構成する関数は，スケジューラ内のポリシに依存する関数よりなる．よって，スケジューラ内の関数において，ポリシに依存する関数とそれ以外の関数が明確になるように整理しなければならない．

・モジュールの管理

モジュールをカーネルに登録した場合，登録対象のモジュールをカーネルに認識させなければならない．そこで，カーネル内にどのようなモジュールの認識機構を設けるかについて検討しなければならない．

・レディキューの操作

レディキューの構成やその操作は，ポリシによって異なる．そこで，モジュールが，レディキューのための領域をどのように確保し管理するかを考えなければならない．

・ポリシが持つ属性

ポリシによっては，様々な属性値を持つことがあり，属性値の種類は，ポリシに依存する．そこで，モジュールが，これらの属性値を格納するための領域をどのように確保し管理するかを考えなければならない．

4 プロトタイプの実装

本研究では，SunOS 上で動作する Minix のカーネルを変更することによってカーネル運用中にポリシの変更を可能とする OS のプロトタイプの実装を試みた．

SunOS 上の Minix は，SunOS から見た場合一つのプロセスとして動作し，1 つのメモリ空間を共有している．

まず，SunOS 上で動作する Minix に関して，以下のように定義する．

・Minix 上で動作するカーネルやプロセスのような実行可能なものをタスクと呼ぶ．

・メモリマネージャ(以下 MM)，ファイルサーバ (以下 FS) といったサーバタスクが存在し，サーバタスクはマイクロカーネル構成におけるシステムサーバと同等のものであると定義する．

4.1 Minix におけるスケジューリング

Minix には，カーネル，サーバタスク，ユーザタスクが存在し，個々に動作する．また，カーネル内にはカーネルタスクと呼ばれる，カーネル内の様々な処理を行なうタスクが存在する．Minix では，タスク単位でスケジューリングが行なわれる．また Minix では，スケジューリングのために，カーネルタスク用のキュー (TASK_Q)，サーバタスク用のキュー (SERVER_Q)，ユーザタスク用のキュー (USER_Q) と，3 本のレディキューを用意している．これらのキューには優先度が付けられており，その順番は高いものから順に

TASK_Q > SERVER_Q > USER_Q

である．

ユーザタスクの場合は，ラウンドロビン (以下 RR) のポリシが適用されている．一方，カーネルタスクやサーバタスクは，作業が終わるまでプロセッサを占有して走り続ける．

本研究では，ユーザタスクのスケジューリングポリシである sched() 関数を置き換えることを目的とする．

4.2 プロトタイプの実装

本研究では，SunOS 上で動作する Minix に最小限の変更を加えることによって，プロトタイプの実装を行なう．本節では，より多くのポリシを実装可能にするための，プロトタイプの実装方針について説明する．

4.2.1 モジュールの実装方針

ポリシを実現するモジュールの各関数の実装方針を以下に定める．

(1) モジュールを構成する全ての関数を，同一のタスク (以下スケジューリングタスク) に実装するようにする．

(2) ポリシの登録操作は，スケジューリングタスクが行なうようにする．

(3) モジュールが属性値を保持するデータ構造は，スケジューリングタスク内に含めるように

する．

(4) ポリシによって，必要とするレディキューの構成は各々異なるが，カーネル内のレディキューの構造を変更することは非常に難しい．よって，プロトタイプの実装では，カーネル内のレディキューには手を加えないこととする．そのかわり，ポリシによっては，独自のレディキューをスケジューリングタスク内に設けて，モジュールが管理するようにする．

4.2.2 モジュールの構成要素など

プロトタイプの実装では，モジュールの構成要素となる関数を以下のようにいくつか用意する．

・カーネル内のデータ構造を認識する関数
・モジュール内の属性値を初期化する関数
・実際にスケジューリングを行なうポリシ関数
・タスクイベント処理関数

ポリシによっては，fork，exit，kill などのユーザタスクに関するイベントに対する処理が必要な場合が存在する．また，複数のレディキューを保有する必要があるポリシを考えた場合，スケジューリングタスク内に設けた独自のレディキューの維持を行なうために，ユーザタスクの実行可能状態への移行や待ち状態への移行などのイベントに対応する処理を実行しなければならない．これらの処理を実現する関数は，ユーザタスクに関するイベントが生起する際にカーネルから関数呼び出しで呼び出される．

カーネル内に，現在使っているスケジューラに関する情報を保持する sched_conf，登録された全てのスケジューラに関する情報を保持する sched_data を用意する．

また，モジュールに対する操作のために実装したシステムコールは登録・削除・切替えの 3 つである．

4.2.3 モジュールに対する操作

ここでは，モジュールの登録，削除，切替え，検索の手順について説明を行なう．

・登録

あるユーザタスクまたはモジュール自身が，登録を行なうシステムコールを発行すると，カーネルは，モジュールの登録を行なう．カーネルは，スケジューリングテーブル sched_data を検索し，既に登録されているか sched_data が全て埋まっている場合は，登録は許可されない．モジュールの登録が許可されると，カーネルは sched_data の登録対象のエントリに，必要な情報を書き込む．登録に成功した場合のみ，モジュールは，カーネル内データ構造の認識を行なう．つまり，カーネルが，モジュール内の特定の領域に，モジュールが必要とするデータ構造や関数のアドレスを設定する．最後にカーネルは，登録に失敗した場合 int 値の 0 を返し，登録が成功した場合，登録された sched_data のインデックスを返す．

・削除

あるユーザタスクまたはモジュール自身が削除を行なうシステムコールを発行すると，カーネルは，モジュールの検査および削除を行なう．カーネルは，削除したいモジュールが，sched_data に登録されていないことや，現在使用中のものかを調べ，該当すればモジュールの削除を許可しない．削除が許可されると，sched_data 内の該当するエントリの登録フラグを 0 に設定する．カーネルは，削除が失敗した場合 int 値の 0 を返し，削除が成功した場合 int 値の 1 を返す．

・切替え

あるユーザタスクまたはモジュール自身が切替えのシステムコールを発行すると，カーネルは，モジュールの切替えの検査を行なう．必要であれば，カーネルはモジュール内の属性値を初期化するため，モジュール内の初期化関数を実行し，切替えを行なう．カーネルは，変更が失敗した場合 int 値の 0 を返し，変更が成功した場合 int 値の 1 を返す．

本研究でのカーネルへの登録・削除・切替えは，カーネルへモジュールのメモリ空間やアドレスを認識させカーネルに利用させることである．メモリイメージはアドレス固定でありリロード等は行なわない．

4.2.4 モジュールの運用

ここでは，モジュールの各関数がカーネルスケジューラ内の関数としてどのように運用されるかということについて説明を行なう．

カーネルから呼び出されたポリシ関数が，ユーザタスクのスケジューリングを行ない，独自のデータ構造を参考にしながら特定のポリシにしたがって，次にプロセッサに割り当てたいタスクをカーネル内の USER_Q の先頭に配置する．

次に，運用されているモジュールにおいて，タスクに関するイベントがどのように処理されるかについて説明する．

(1) カーネルからイベント処理関数が呼び出される．

(2) イベント処理関数は，タスクに関するイベントの処理を行なう．

カーネルから呼び出されたイベント処理関数は，引数で指定されたイベントの種類やタスク情報を元にして，独自のデータ構造の更新などの処理を行なう．

5 スケジューリングポリシの実装

本章では，プロトタイプ上に実装したスケジューリングポリシについての説明を行なう．

スケジューリングポリシは，カーネル内部に実装されているものとユーザタスク内に実装されているものの 2 種類がある．カーネル内部には RR と FIFO が，ユーザタスク内では，RR, FIFO, 優先度付きスケジューラ，タイムスライスの変更可能な RR スケジューラ，複数のキューをもつスケジューラと，計 7 つのポリシが存在する．ユーザタスク内のものは登録さえすれば，7 つ全てが自由に切替え可能である．

RR と FIFO に関しては，カーネル内のものもユーザタスク内のものもほぼ同一の動作をする．

FIFO では，ポリシ関数として，何も行なわない．これにより，疑似的に FIFO のスケジューリングが実現される．

優先度付きのスケジューリングポリシでは，最も優先度の大きいユーザタスクを，プロセッサに割り当てる．もし，そのようなユーザタスクが複数存在する時は，その間で順々に実行が行なわれる．このポリシの実装では，属性値を保持するために，スケジューリングタスク内にデータ構造を用意し，タスク番号で参照できるようにする．属性値である優先度は 1 以上の整数をとり，初期値を 1 とする．

タイムスライスの変更可能な RR スケジューリングポリシは，RR と同じ方法で行なわれるが，各ユーザタスク毎に設定されたタイムスライスでタスクの実行が行なわれる．このポリシの実装では，タイムスライスは都合上，常に単位タイムスライス (300ms) の整数倍とする．属性値であるタイムスライス値にはこの倍率が格納され，初期値は 1 である．

複数のキューを持つスケジューリングポリシでは，ユーザタスクのためのレディキューが必要であり，スケジューリングタスク内に独自の構造に基づいた複数のレディキューを設ける．また，各ユーザタスクごとに属性値として所属レディキューに関する情報を持つ．

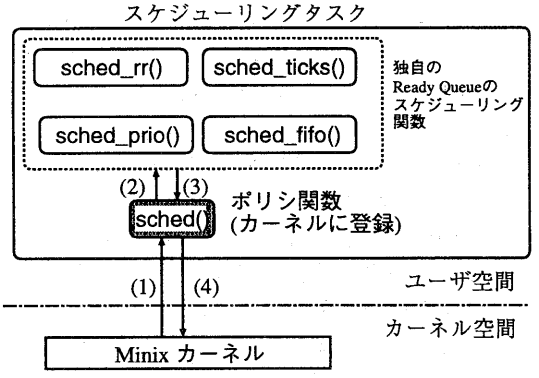

図 4: ポリシ関数の実行

複数のキューをもつスケジューラのスケジューリングの手順を以下に示す．

(1) カーネルが当該関数を呼び出す (図 4の (1))．

(2) 当該関数は，スケジューリングを行なうレディキューを順番に選択する．

(3) 選択されたレディキューを操作する．

ポリシには，タイムスライスの変更可能なも

の (sched_ticks()), RR (sched_rr()), FIFO(sched_fifo()), 優先度付き (sched_prio()) と, 4 つが存在する. 上記の各関数では, 指定されたレディキューの中からユーザタスクを選択し, レディキューの先頭に配置し, タスク番号をスケジューリング関数に返す (図 4 の (3)).

(4) (3) で得られたタスクがカーネルの USER_Q の先頭に配置されるように, カーネル内の USER_Q を操作する.

(5) カーネルに戻る (図 4 の (4)).

6 性能評価

本章では, 前章で実装したポリシを使用して, 図 5で示されるテストプログラムの実行時間を計測した.

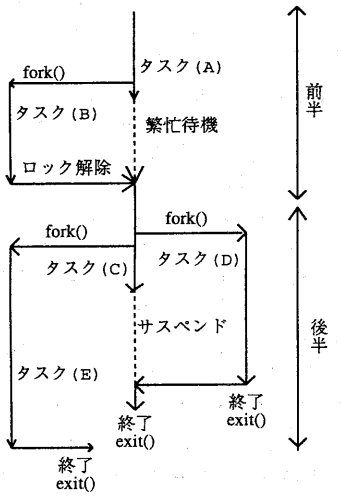

図 5: テストプログラムの実行

図 5 で示されるテストプログラム, 以下の手順で実行される. 以下, ループの回数は適当と思われる値を選んだ.

(1) ポリシ決定のシステムコールを発行する.

(2) タスク (B) を fork する. タスク (B) とタスク (A) は, ポリシに特有の属性値を設定するシステムコールを発行する. また, タスク (B) はある共有変数をロックする.

表 1: テストプログラムの実行パターンと測定結果

	ポリシ	測定時間 (秒)
(a)	RR → RR	147
(b)	FIFO → FIFO	198
(c)	FIFO → RR	94
(d)	FIFO → 優先度付き	73
(e)	タイムスライス → FIFO	356

(3) タスク (B) のプログラムは, 浮動小数点数の変数に定数を加算するルーチンを 5000 万回実行する. この間, タスク (A) は, (2) での共有変数のロック解除を待ち繁忙待機を行なう.

(4) タスク (B) は, ルーチンの実行を終えると, タスク (A) 内のロック変数を解除して終了する.

(5) タスク (D) を fork する. 浮動小数点数の変数に定数を加算するルーチンを 2000 万回実行する. 必要であれば, タスク (D) でポリシに特有の属性値を設定するシステムコールを発行する.

(6) タスク (E) を fork する. タスク (D) と同じ計算ルーチンを, タスク (D) の場合の 6 倍すなわち 1 億 2000 万回実行する. 必要であれば, タスク (E) でポリシに特有の属性値を設定するシステムコールを発行する.

(7) タスク (C) は, ブロックされて待ち状態に入る. いずれかの子タスクの終了時までタスク (C) の待ち状態が続く.

(8) タスク (D), (E) のいずれかが終了すると, タスク (C) は待ち状態から抜け, 実行を終了する.

プログラムの後半部が終了した時点で, (C) の方で実行時間を出力して全プログラムは終了する. 必要があれば, プログラムの前半と後半の間にポリシ入れ換えのシステムコールを発行する.

以下, スケジューリングポリシの変更を考慮して, いくつかのパターンでテストプログラムを実行し, 実行時間を測定・比較する.

表 1に各実行パターンとその測定結果を示す.

(d) ではタスク (C) とタスク (D) は同じ優先度で，タスク (E) はそれより低くする．また，(e) ではタイムスライスは，タスク (A) がタスク (B) の 3 倍とする．

理論的には，以下の実行状態が最も実行時間を短くできるはずである．

・タスク (B) がプロセッサを占有しタスク (A) に実行権限を渡さない

・タスク (D) がプロセッサを占有した状態でタスク (C) にもタスク (E) にも実行権限を渡さずに終了し，タスク (C) に実行権限を移す．

(d) の状態が最も実行時間が短い．これは，最も理想的な状態でプログラムが動作されているからである．前半は (b) の前半と同じで，後半は実行時間の短いタスク (D) がプロセッサを占有し実行終了した後，タスク (E) に実行権限が移らずタスク (C) に実行権限が移るからである．

(e) の状態が最も実行結果が悪い．これは，前半に実行の必要のないタスク (A) に多くの実行時間が割り当てられ，後半では，タスク (D), (E) の両方が終了するまでプログラムが終了できないからである．

複数のキューを持つスケジューラで上記と同様の実行状態で実行させても同様の実行結果が得られる．

7 おわりに

本研究では，スケジューリングポリシの動的変更が可能な OS の構成法の提案とプロトタイプの実装を行なった．また，プロトタイプ上で様々なスケジューリングポリシの実装を行ない，テストプログラムを用いて，性能評価を行なった．その結果，プログラムの特性に応じてスケジューリングポリシを選択することにより，ポリシを変更しない場合と比較してプログラムの実行時間が短縮されることが確認された．

今後の課題としては，以下のようなことが挙げられる．

・より多様なスケジューリングポリシの実現

・メモリ空間の保護

今回の実装では，簡単のため，カーネルとプロセスの間メモリ保護については特に考慮しなかった．よって，メモリ保護が適用される場合のモジュールのカーネル内への取り込みをどのように行なうかについて考慮しなければならない．

最終的には，独自で決定した仕様のカーネル上でのより多様なスケジューリングポリシの実装を目指す．

参考文献

[1] M. Rozier et al. : Overview of the CHORUS distributed operating system, *Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp.39–69 (1992).

[2] Brian N.Bershad et al.: SPIN - An Extensible Microkernel for Application-specific Operating System Services, *Proc. SIGOPS European Workshop*, pp.74–77 (1994).

[3] J. Mitchell et al.: An Overview of the Spring System, *Proc. Compcon Spring 1994*, pp.122–131 (1994).

[4] Y. Yokote : The Apertos reflective operating system: the concept and its implementation, *SIGPLAN Not. (USA), Vol.27, No.10*, pp.414–434 (1992).

[5] 柏木 一彦他：動的保護が可能な動的構築機能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，情報処理学会論文誌，第 40 巻，第 6 号，pp.2618–2634(1999).

[6] 藤原 真：カーネルスケジューラの動的変更が可能なオペレーティングシステムの構成法，奈良先端科学技術大学院大学修士論文 (1999).

[7] Andrew S.Tanenbaum : *Operating System - Design and Implementation*, Prentice Hall (1987).