システムソフトウェアと
オペレーティング・システム 70−14
（１９９５．８．２３）

異なるスケジューラの共存制御法

村上 大介　谷口 秀夫　牛島 和夫
九州大学工学部
〒８１２−８１　福岡県福岡市東区箱崎６−１０−１

計算機のハードウェアの性能向上はソフトウェア処理の速度を向上させた。しかし、ハードウェアの性能向上により幾つかの問題が生じている。例えば、性能の異なるプロセッサが存在するため、プロセッサ性能に依存するソフトウェア処理は、プロセッサ機種毎の調整を必要としている。また、高過ぎるプロセッサ性能のため、人間とのインタフェースを提供しているソフトウェア処理は、人間の時間感覚に合わせた速度調整が必要になっている。本稿では、要求するプロセッサ性能を提供するスケジュール法と既存のスケジュール法の共存制御法を提案する。さらに、実際に試作することにより、提案方式を評価、考察する。

Mechanism for Coexistence of two different Schedulers

Daisuke MURAKAMI, Hideo TANIGUCHI, and Kazuo USHIJIMA
murakami@csce.kyushu-u.ac.jp
tani@csce.kyushu-u.ac.jp
ushijima@csce.kyushu-u.ac.jp

Faculty of Engineering, Kyushu University
6-10-1, Hakozaki, Higashi-ku, Hukuoka, Hukuoka, 812-81 Japan

Increase of computer hardware performance speeds up software processing. But there are some problems by that. For example, software processing depended on processor performance need to adjust processing to type of processor because processors have different performance one another. And software processing which implement human interface need to adjust processing speed to human sense independent of processor performance. In this paper, we propose a mechanism for coexistence of two different schedulers, one is the scheduler which gives required processor performance and another is the scheduler controlled by process priority. And we describe implementation and evaluation of this mechanism.

1 はじめに

ハードウェアの性能による影響を抑え、サービスが必要とする処理時間を保証できるオペレーティングシステム（以降、ＯＳと略す）の研究を進めている[1]。特に、プロセッサの性能とＯＳのスケジュール法とは、ソフトウェアの処理時間に深く関係する。文献[1]では、サービス処理時間を保証する性能保証スケジュール法を提案し、設計している。一方、既存スケジュール法は、プロセッサを有効利用する観点から設計されている。それゆえ、性能保証スケジュール法と既存スケジュール法の共存は、種々の計算機利用を可能にするために有効である。

本稿では、２つのスケジュール法を共存制御する際の課題を明らかにし、各々に対する対処法を提案する。さらに、共存制御の方式を実現し、評価する。

2 既存スケジュール法と性能保証スケジュール法

2.1 既存スケジュール法

既存スケジュール法として、ＢＳＤ／３８６ＵＮＩＸ（１．０版）のスケジュール法を考える。以降、プロセススケジューリングの仕組み、またプロセス切替の契機について説明する。これにより、２つのスケジュール法を共存させる際のプログラム構造を探求する。

スケジューラに関連するプログラム構造を図１に示し、スケジューラやディスパッチャの呼出契機と呼出方法を表１に示す。

スケジューラが明示的に呼び出されるのは立ち上げ時のみである。しかし、スケジューラの処理の最後では、自ら１秒後の再呼出を設定している。こうして、１秒毎にソフトウェア割込として呼び出される。

ディスパッチャは休眠（停止、終了）、覚醒、その他多くのモジュールと関わっている。それらのモジュールがディスパッチャを呼び出す方法には、フラグ要求による間接呼出と、明示的な直接呼出とがある。スケジューリングやプロセス覚醒などによってプロセス優先度が変動したなら、フラグをセットすることでプロセス切替が必要である旨を示す。実際にディスパッチャが呼び出されるのは、ＯＳ側の処理が終った後、カーネルモードからユーザモードへ戻る際のことになる。一方、プロセスの休眠や停止、終了などによってプロセッサが放棄されたなら、即座のプロセス切替が必要になる。従って、これらのモジュールはディスパッチャを直接呼び出す。

スケジューラは、全てのプロセスについて、プロセッサ使用率を計算し、それを元にプロセス優先度を再算出する。それによって現在実行中のプロセスより高い優先度を持つプロセスが生じたなら、プロセス切替を要求するフラグをセットする。また、優先度が変わったプロセスは、レディーキュー内の正しい位置へつなぎ直す。最後に、タイムアウトキューに、１秒後スケジューラを呼び出すようセットし、処理を終る。

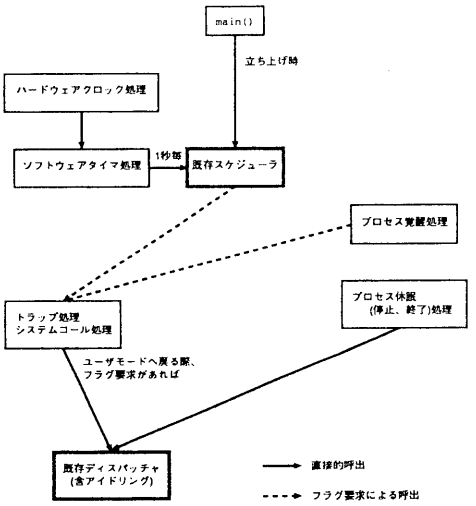


図１ 既存のスケジューラ周りの構造

表１ スケジューラ、ディスパッチャの呼出契機と呼出方法

	呼出契機	呼出方法
スケジューラ	立ち上げ時	直接
	１秒毎	ソフトウェア割込み
ディスパッチャ	プロセス優先度順位の変動時	フラグ要求
	プロセスの休眠、停止、終了時	直接

2.2 性能保証スケジュール法

性能保証スケジュール法[1]はプロセッサ性能を可変に提供するスケジュール法である。この方法について、

（１）プロセスへのプロセッサの割当方式
（２）プロセスがプロセッサを途中放棄した時の対処方式

を説明する。

2.2.1 プロセッサ割当方式

要求されたプロセッサ性能を提供するため、サービス処理の実行と停止を適度の間隔で繰り返す。具体的には、プロセッサの実行を一定の単位時間（以降、タイムスロットと呼ぶ）に分割し、性能保証を行なうべきプロセス（以降、性能保証プロセスと呼ぶ）に対

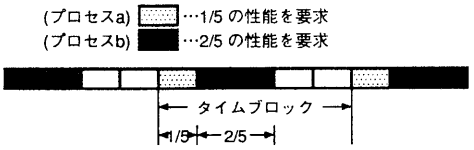


図2 ブロック内割当方式

しては、タイムスロット割当の割合を制御する。

ここでは、連続するN個のタイムスロット（以降、タイムブロックと呼ぶ）内で、要求性能に応じた個数のタイムスロットを割り当てる制御方式（以降、ブロック内割当方式と呼ぶ）を説明する。この方式の様子を図2に示す。図2では、タイムスロットを5個まとめてタイムブロックとし、要求に応じプロセスaには1個、プロセスbには2個のタイムスロットを割り当てている。これにより、プロセスaに$\frac{1}{5}$のプロセッサ性能を、プロセスbに$\frac{2}{5}$のプロセッサ性能を提供している。

ブロック内割当方式は、タイムブロック内のタイムスロット数を越えた、非常に低い性能の提供ができないという欠点があるものの、以下の利点がある。

（1）割当の制御が簡単である。

（2）プロセス切替の発生を少なくできる。

2.2.2 プロセッサ途中放棄への対処

プロセスは、タイムスロット途中で（事象待ちや終了のため）プロセッサ割当を放棄する可能性がある。そのような場合の対処として、2つの方式が考えられる。2つの方式を表2に示す。1つは、タイムスロットの残り時間は、性能保証を行なわないプロセス（以降、性能非保証プロセスと呼ぶ）に割り当てる方式（以降、（方式a）と呼ぶ）である。性能非保証プロセスがなければ、アイドリングさせる。（方式a）の長所として、

（1）サービス処理の均一性を損なわない。

（2）性能非保証プロセスと共存する場合、プロセッサを有効利用できる。

（3）制御が簡単である。

点が挙げられる。しかし短所として、

（A）性能保証プロセスしかない場合は、プロセッサ利用に無駄が生じる。

点が挙げられる。もう1つは、タイムスロットの境界を再調整し、次のプロセスのタイムスロットへ移る方式（以降、（方式b）と呼ぶ）である。（方式b）の長所として、

（1）プロセッサを有効利用できる。

点が挙げられる。しかし短所として、

（A）処理の均一性が損なわれる恐れがある。

（B）タイムスロット境界の調整処理とそれを可能にする時刻管理とが必要になり、制御が複雑になる。

表2 プロセッサ途中放棄への対処方式

方式	長所	短所
（方式a）…タイムスロットの残り時間は、性能非保証プロセスに割り当てる。なければ、アイドリングさせる。	●サービス処理の均一性の保存 ●性能非保証プロセスと共存時の、プロセッサ利用効率の高さ ●制御の簡便さ	●性能保証プロセスのみの場合の、プロセッサ利用効率の低さ
（方式b）…タイムスロットの境界を再調整し、次のプロセスのタイムスロットへ移る。	●プロセッサ利用効率の高さ	●処理の均一性の破壊 ●タイムスロット境界調整と時刻管理とによる制御の複雑さ

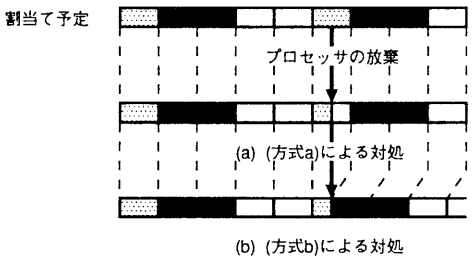


図3 プロセッサ途中放棄への対処方式の比較

点が挙げられる。2つの方式の比較を図3に示す。図3（a）は（方式a）の例であり、図3（b）は（方式b）の例である。図3から分かる通り、（方式a）は一見して単純な制御法である。また、処理の均一性も保たれている。しかし、放棄されたタイムスロットの残余がアイドリングされており、プロセッサ利用効率から見れば無駄である。一方、（方式b）は、（方式a）に比して明らかに無駄が少ない。しかし、そのためにタイムスロット境界を変動させてしまっており、処理の均一性も乱れている。

本研究では、以下の2つの理由から（方式a）を採用する。

（1）制御が簡単である。

（2）性能非保証プロセスとの共存に適している。

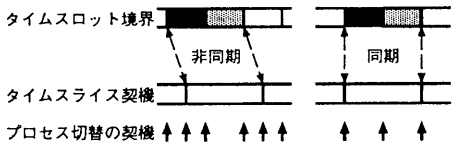
3 共存制御における課題と対処

両スケジュール法を共存させるため、その制御方式を確立し、実現方式を明らかにする。具体的には、

（1）プロセス切替契機の設定
（2）スケジュール法の呼出
　（A）呼び出すスケジュール法の選択
　（B）既存スケジュール法の呼出
について対処を説明する。

表3 プロセス切替の契機

起因となるスケジュール法	プロセス切替の要因	発生の性質
性能保証	タイムスロット境界	定期的
既存	タイムスライス契機	
	プロセスの覚醒	不定期的
	プロセスの休眠	



(a) 同期していない場合 (b) 同期している場合

図4 タイムスロット境界とタイムスライス契機の関係

3.1 プロセス切替契機の設定

サービス処理時間を保証し、かつプロセッサを有効利用する観点から、プロセス切替回数を削減することが重要である。そのため、プロセス切替の契機となるタイムスロット境界の設定においては、既存スケジュール法との共存により、プロセス切替の契機が増加しないような配慮が必要である。

両スケジュール法の共存状態でのプロセス切替の契機を、表3に示す。不定期的な要因によるプロセス切替の発生はプロセスの処理内容に依存するため、それを制御することは難しい。一方、定期的に発生するプロセス切替要因については、発生を同期させるように制御できる。

タイムスロット境界とタイムスライス契機の関係を図4に示す。図4（a）は、タイムスロット境界とタイムスライス契機が同期していない例であり、図4（b）はタイムスロット境界とタイムスライス契機が同期している例である。図4から判るように、タイムスロット境界とタイムスライス契機を同期させることで、プロセス切替の契機を削減できる。

3.2 スケジュール法の呼出

プロセス切替時には、呼び出すスケジュール法の選択と、既存スケジュール法の呼出が課題となる。

表4 スケジュール法の選択

プロセス切替の要因	次タイムスロットの割当状態	性能保証状態か否か	選択するスケジュール法
タイムスロット境界	割当有		性能保証
	割当無		既存
プロセスの覚醒		Yes	性能保証
		No	既存
プロセスの休眠		Yes	既存
		No	

3.2.1 呼び出すスケジュール法の選択

プロセス切替時のスケジュール法の選択は、プロセス切替の要因に依存する。プロセス切替の要因と次タイムスロットの状態、現プロセスの状態、そして選択するスケジュール法の関係を表4にまとめる。

プロセス切替の要因がタイムスロット境界の場合は、次タイムスロットに性能保証プロセスが割り当てられているか否かにより、スケジュール法を選択する。割当があれば性能保証スケジュール法を呼び出し、割当がなければ既存スケジュール法を呼び出す。一方、プロセス切替の要因がプロセスの覚醒や休眠の場合は、現タイムスロットが性能保証状態か否か（以降、割り当てられた状態を性能保証状態、割り当てられていない状態を性能非保証状態と呼ぶ）により、スケジュール法を選択する。性能保証状態ならば性能保証スケジュール法を呼び出し、性能非保証状態ならば既存スケジュール法を呼び出す。

3.2.2 既存スケジュール法の呼出

既存スケジュール法は、実行可状態にあるプロセスの中から、プロセス優先度に基づいて次に実行するプロセスを決定する。しかし性能保証スケジュール法と共存する場合、既存スケジュール法が実行すべきプロセスは性能非保証プロセスのみでなければならないので、性能保証プロセスは実行可状態であっても実行してはならない。例えば、性能保証プロセスの管理を性能非保証プロセスとは別に行なえば、既存スケジューラが性能保証プロセスにプロセッサを割り当てることはない。しかし、共存制御による処理を局所化するため、既存スケジュール法の変更は避けたい。したがって、既存スケジュール法を呼び出す前には、性能保証プロセスを実行することがないような対処が必要になる。例えば、性能保証プロセスのプロセス優先度を最下位まで落としたり、性能非保証プロセスが見つからない時には、アイドリング処理を行なったりする。

3．2．1項と3．2．2項に示した内容に従い、1つの処理モジュールを導入する。これを転轍機関数と名付ける。

転轍機関数は、プロセス切替の要因と次タイムスロットの状態、現プロセスの状態を勘案し、呼び出すスケジュール法を選択する。また、既存スケジュール法を呼び出す際には、あらかじめレディーキューを検索し、性能非保証プロセスの存在を確認しておく。性能非保証プロセスより優先度の高い性能保証プロセスは、優先度が最下位のキューへつなぎ替えてしまう。そして、もし性能非保証プロセスが見つからない時には、プロセッサをアイドリングさせる。アイドリングから復活したなら、スケジュール法の選択から再開する。

4 実現方式

4.1 基本方針

作成の工数、他ＯＳへの適応性を考慮し、新機能はモジュール化する。具体的には、以下の３つの方針に従う。

（方針ａ）スケジューラやディスパッチャの変更を極力抑える。

（方針ｂ）スケジューラやディスパッチャを呼び出す関数の変更を抑える。

（方針ｃ）既存の他のプロセス管理機能のインタフェースを保存する。

4.2 提供形式

性能を要求するシステムコールの形式を、表５に示す。

表5 システムコールの形式

形式	`set_performance(n)` `long n`
内容	•n > 0 の場合、実プロセッサ性能の n / 20 の性能でプロセスを走行させる。 •n = 0 の場合、現在提供されている性能を返す。 •n < 0 の場合、既存のスケジュール制御で走行させる。

4.3 プログラム構造

２章から３章までの検討内容を以下にまとめる。

（1）プロセッサのプロセスへの割当には、ブロック内割当方式を用いる。

（2）プロセッサの途中放棄への対処には、タイムスロットの残余に性能非保証プロセスを割り当てる方式を用いる。

（3）新機能はモジュール化する。

（4）プロセス切替契機の削減のため、タイムスロット境界はタイムスライス契機と同期するよう設定する。

（5）プロセス切替時のスケジュール法の呼出を行なうため、転轍機関数を導入する。

上記の各項に対応する実現時の対処内容を以下に示す。

（1）タイムブロックを示す配列 `time_block[]` と、タイムブロック内の現在位置を示す変数 `time_index` を導入する。

（2）プロセッサが途中放棄された場合、既存スケジュール法を呼び出しプロセス切替を行なう。

（3－1）既存のスケジューラの変更を避けるため、性能保証のためのスケジュール法を独立関数 `new_sched_cpu()` として実装する。

（3－2）既存のディスパッチャの変更も極力抑え、関数名だけを `old_swtch()` に変更して既存スケジュール法の呼出に用いる。

（3－3）性能保証スケジュール法の呼出を行なう新関数 `new_swtch()` を実装する。

（3－4）ディスパッチャを呼び出す関数群の変更を避け、かつ他のプロセス管理機能のインタフェースを保存するため、転轍器関数の名前には既存ディスパッチャの変更以前の関数名 `swtch()` を用いる。

（4）タイムスロット長は、タイムスライス時間間隔の $\frac{1}{100}$（$\frac{1}{100}$秒）とする。

（5）転轍機関数 `swtch()` を実装し、呼出時にスケジュール法の選択に用いるフラグ `new_want_resched` を導入する。

新モジュールを加えたプログラム間の関係を、図５に示す。また、新モジュールのインタフェースを表６に示す。図５では、図１に比べ、網かけの処理が新モジュールである。性能保証スケジューラは、既存スケジューラと異なり、ハードウェアクロック処理から呼び出される。これはタイムスロット境界がやってき

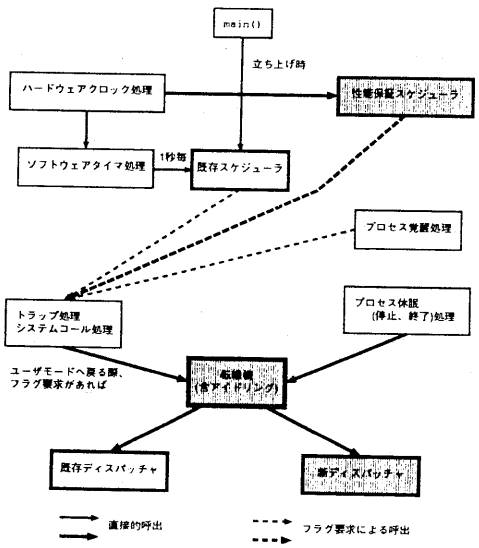


図5 新スケジューラ周りの構造

表6 新モジュールのインタフェース

関数名	引数	概要
new_schedcpu()	なし	タイムスロット毎に性能保証プロセスのスケジューリングを行なう。
old_swtch()	なし	通常のスケジューリングに従って、プロセスコンテクストを切替える。旧swtch()と全く同じ動作をする。
new_swtch()	なし	性能保証プロセスへ、プロセスコンテクストを切り替える。
swtch()	なし	転轍機関数。new_swtchとold_swtchのいずれを呼ぶべきか選択する。old_swtchを呼ぶ場合、性能非保証プロセスの存在を確認し、性能保証プロセスが実行されないような対処を施す。

た時、即座にプロセッサを確保するためである。性能保証スケジューラからのプロセス切替要求は、既存スケジューラと同じくフラグ要求による。プロセス切替は、常に転轍機関数を経由するように変更した。ここで、呼び出すスケジューリング法を選択し、ふさわしいディスパッチャへ処理を移す。

5 評価と考察

既存スケジュール法と性能保証スケジュール法との共存制御方式を実現したOS（以降、試作OSと呼ぶ）を作成し、以下の項目を評価した。

（A）共存制御による処理時間のオーバヘッド

（B）性能保証スケジュール法が提供する性能の精度

5.1 環境条件

評価プログラム（以降、TPと略す）には、プロセッサ処理と入出力処理を交互に行なう処理を用いた。具体的には、プロセッサ処理はループ処理、入出力処理は磁気ディスクからの入力処理とした。評価プログラムの処理時間は、単独走行時で3～7秒（処理内容によって異なる）とした。

TPの処理時間の測定にはtimes()システムコールを用い、測定単位は$\frac{1}{60}$秒である。測定は、シングルユーザモードでそれぞれ20回行ない、処理時間はその最大値、最小値を除いた残りの値の平均値とした。

5.2 共存制御によるオーバヘッド

スケジュール法の共存制御によるオーバヘッドを明らかにした。具体的には、TPを既存OS上で実行したときの処理時間と、TPを試作OS上で性能保証を行なわずに実行したときの処理時間とを比較した。オーバヘッド率は、

$$\left(\frac{\text{試作OS上での処理時間}}{\text{既存OS上での処理時間}} - 1\right) \times 100 \quad (\%)$$

で計算した。

5.2.1 処理内容によるオーバヘッドの変化

TPのプロセッサ処理と入出力処理の割合を変化させて、単独で実行し、処理時間を測定した。TPの全処理に対する入出力処理の比率とオーバヘッド率の関係を図6に示す。実線は測定値である。一点鎖線は測定値の平均値である。点線は、最小自乗法を用いて算出した、測定値の1次近似である。

図6より、以下のことがわかる。

（1）オーバヘッド率は1．8％以下であり、平均オーバヘッド率は0．75％である。

（2）入出力処理の比率の増加とともに、測定値が大きく変動している。これは、以下の理由による。

処理内容に入出力処理の占める比率が大きくなると、I／O終了割込みとタイマ割込みが衝突する可能性が高くなる。I／O終了割込みとタイマ割込みが衝突した場合、タイマ割込みの処理の方が優先され、入出力終了割込みの処理は待たされる。そのため、それだけ処理時間が長くなる。一方、入出力処理の比率が大きくなれば、I／Oを処理している待ち時間の間にタイマ割込みが起きる可能性も高くなる。I／O待ちの間にタイマ割込み処

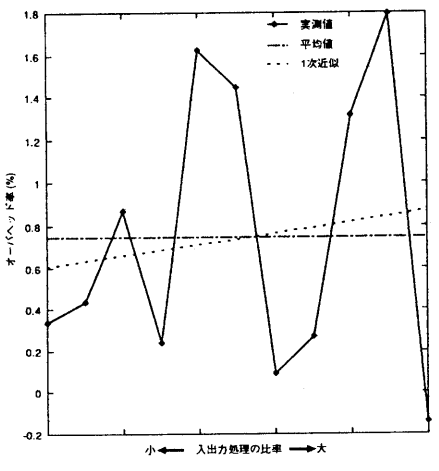


図6 入出力処理の比率とオーバヘッド率の関係

理が実行されれば、処理時間は短くなる。そのため、入出力処理の微妙なタイミングによって処理時間は変動する。従って、入出力処理の比率の増加とともに、処理時間の変動の度合も大きくなり、それに伴ってオーバヘッド率も変動する。

（3）測定値の１次近似（点線）より、処理内容における入出力処理の比率の増加とともにオーバヘッド率が上がる傾向がある。これは、以下の理由による。

処理内容に入出力処理の占める比率が大きくなると、プロセスの休眠、覚醒が頻発し、プロセス切替の契機が増大する。図５より、試作ＯＳは既存ＯＳと比べて、転轍機関数の処理量だけプロセス切替に時間を要する。従って、入出力処理の比率の増加とともに、オーバヘッド率も上がる。

5.2.2 プロセス数によるオーバヘッドの変化

ＴＰを単独で実行するのではなく、ＴＰと同様の処理を行なうプログラムも同時に実行した。ＴＰを含めた実行プロセスの総数を変化させて、処理時間を測定した。実行するプロセスの処理内容は、プロセッサ処理のみとした。プロセス数によるオーバヘッド率の変化を図７に示す。実線は測定値である。一点鎖線は測定値の平均値である。点線は、最小自乗法を用いて算出した、測定値の１次近似である。

図７より、以下のことがわかる。

（1）平均オーバヘッド率は０．８９％である。

（2）プロセス数の増加とともに、測定値が大きく大きく変動している。これは、以下の理由による。

もし、プロセスが処理を終える直前にタイムスライスを使い切ったなら、残りがわずかな処理であっても、再度タイムスライスを獲得するまで待たなければならない。そのため、処理の微妙なタイミングによって、処理時間は変動する。タイムスライス再獲得までの待ち時間は、プロセス数が増えるほど長くなる。従って、プロセス数の増加とともに処理時間の変動の度合も大きくなり、それに伴ってオーバヘッド率も変動する。

（3）測定値の１次近似（点線）より、同時に実行するプロセス数が増してもオーバヘッド率には変化の傾向はほとんどあらわれない。これは、同時に走行するプロセス数が増しても、プロセス１つあたりのプロセス切替契機は増えないからである。

5.3 提供性能の精度

性能保証スケジュール法が提供する性能の精度を明らかにした。具体的には、予想処理時間とＴＰを性能保証を行なって実行したときの処理時間を比較した。ここで予想処理時間は、試作ＯＳ上でＴＰを性能保証を行なわずに単独実行したときの処理時間を基準として、以下の式で求めた。

$$\frac{\text{性能非保証時の処理時間}}{\text{要求性能（％）}} \times 100$$

また、ＴＰの処理内容は、全てプロセッサ処理とした。

5.3.1 提供性能の精度

ＴＰが要求する性能を５％から１００％まで５％単位で変化させて、単独で実行させた。これにより、要求性能と実際に提供された性能の関係を明らかにした。要求性能と提供性能の関係を図８に示す。

図８から、要求性能と提供性能との差は±０．５

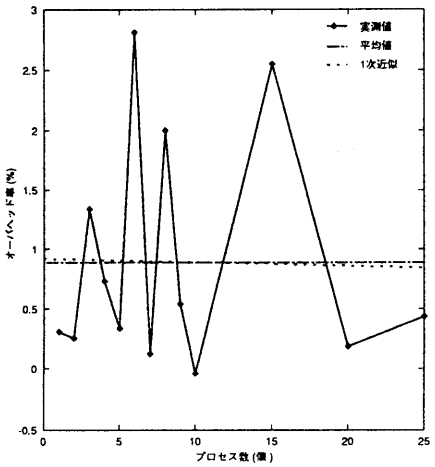


図７ プロセス数によるオーバヘッド率の変化

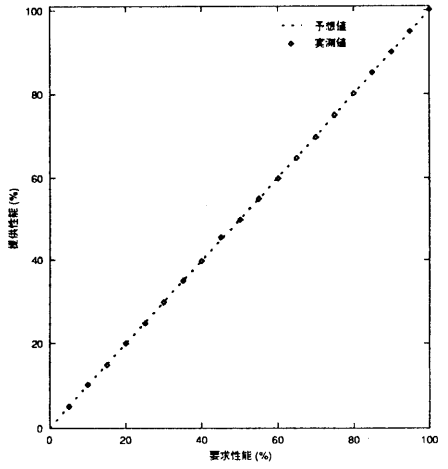


図８ 要求性能と性能保証の関係

%以下であることがわかる。

5.3.2 プロセス数による精度の変化

ＴＰを単独で実行するのではなく、ＴＰと同様の処理を行なうプログラムと同時に実行した。ＴＰは性能保証プロセスとして実行し、同時に実行する他のプロセスは全て性能非保証プロセスとして実行した。実行する性能非保証プロセスの処理内容は、プロセッサ処理のみとした。ＴＰを含めた実行プロセスの総数を変化させた。これらにより、性能非保証プロセス数を変化させた時の提供性能への影響を明らかにした。ＴＰの要求性能が５％の場合について、プロセスの総数と提供性能の関係を図９に示す。

図９より、以下のことがわかる。

（１）要求性能と提供された性能との差は±０．５％以下である。

（２）同時に走行する性能非保証プロセスの数の増加とともに、要求性能が低下する傾向がある。これは、以下の理由と推察する。

２．１節で述べたように、スケジューラは定期的（１秒毎）に全てのプロセスについてプロセス優先度を再算出する。そのため、スケジュール処理に要する時間は同時に実行されるプロセス数が増えると増大する。プロセスの処理時間にはスケジュール処理にかかる時間も含まれている。従って、処理時間が長大化し、提供性能が低下する。

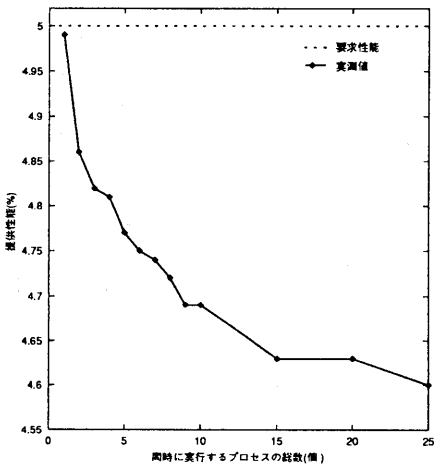


図９ プロセス数による提供性能の変化（５％）

6 おわりに

性能保証スケジュール法と既存スケジュール法を共存制御する方式について報告した。

初めに、既存スケジュール法の例として、ＢＳＤ／３８６ＵＮＩＸ（１．０版）のスケジュール法の動作を示した。次に、性能保証スケジュール法について、共存制御の観点から、プロセッサの割当方式はブロック内割当方式、プロセッサの途中放棄への対処方式は性能非保証プロセスを走行させる方式が、好ましいことを示した。さらに、両スケジュール法の共存制御法を提案した。具体的には、プロセス切替の契機を削減するためには、タイムスロット境界をタイムスライス契機と同期させることが有効であることを示した。また、プロセス切替の際のスケジュール法の呼出について、スケジュール法の選択法と、既存スケジュール法の呼出法を示し、転轍機関数の導入を提案した。また、実現時の対処内容を述べた。最後に、提案した共存制御方式を実現したＯＳを試作し、共存制御方式の評価を行なった。共存制御によるオーバヘッドは、平均して１％以下であり、十分小さい。また、要求性能と提供性能の差は、０．５％以下と、非常に高い。

残された課題として、処理内容による性能保証の精度への影響、またプロセスの生成、消滅が多い状況での対処方法がある。

＜参考文献＞

[1] 谷口秀夫，可変なプロセッサ性能を提供するスケジュール法，コンピュータシステム・シンポジウム，pp.63-70
(1994).