計算機アーキテクチャ　83－18
（１９９０．７．１８）

粒度を考慮したマルチプロセッサの資源管理

森山 孝男　根岸 康　渦原 茂　松本 尚
日本アイ・ビー・エム株式会社 東京基礎研究所

マルチプロセッサ上で、粒度が細かく密に協調並列動作するプログラムの存在を考慮に入れたプロセッサ資源の管理法およびスケジューリングの戦略について検討する。実行コードの生成過程から静的に判断される粒度の情報を利用して、並列動作するプログラムを４種類のカテゴリーに分類する。この分類を利用して管理ならびにスケジューリングを行うため、実プロセッサを割り当てる単位であるプロセスに要求プロセッサ数とモードという属性を付加する。さらに、ユーザとカーネルの２階層のスケジューリング方式で使用できるカーネルによるプリエンプション方式を提案する。この方式はプリエンプションの予告を伴わず、従来の時限爆弾方式よりも優れている。

A Multiprocessor Resource Management Scheme which Considers Program Grain Size

Takao Moriyama　Yasushi Negishi　Shigeru Uzuhara　Takashi Matsumoto

IBM Research, Tokyo Research Laboratory, IBM Japan, Ltd.

5-19, Sanbancho, Chiyoda-ku, Tokyo 102, JAPAN

We examine the strategies for scheduling and resource allocation in multiprocessor operating systems, in the presense of fine-grain and tightly-cooperating programs. First, concurrent programs are classified into four categories based on the grain-size data acquired from the code generation phase. Scheduling and resource allocation based on this data requires that two attributes, mode and number processors demanded, be added to a process, the unit to which physical processors are allocated. Also proposed is a kernel preemption mechanism which can be used in a two-level (user/kernel) scheduler. Our mechanism is superior to the previous time-bomb mechanism in that the preemption notification is not necessary.

1 始めに

マルチプロセッサ上の資源管理を並列処理を意識しておこなう一方式を提案する。本稿では主に実プロセッサ資源の管理法とスケジューリング方式について議論する。簡単のためUMA (Uniform Memory Access) のマルチプロセッサをモデルとして議論を進める。また、マルチプロセッサはプログラムの持つ局所性を利用することを前提とした同質な従来型のプロセッサで構成されるものと仮定する。このマルチプロセッサ上で、粒度の細かい並列処理を行なう場合、処理間で頻繁に要求される通信および同期を効率良く処理する必要がある。今、ここで仮定しているプロセッサではコンテクスト切り換えのコストが大きく、オーバーヘッドを減らすためには細粒度の処理はプロセッサに対応する処理列（シュレッドやスレッド）に静的にスケジューリングされていなければならない。

これらの処理列間では頻繁に同期が要求されるので、これらは同時にまとまって実プロセッサに割り付けられることが望ましい。つまり、粒度の細かい並列処理ほど個別のプロセッサではなく、複数のプロセッサからなるプロセッサ群の割り当てが必要である。特に、並列化コンパイラ [1, 2] がコードを生成する場合や粒度の細かい並列型言語 [3] からコンパイラでプロセッサの実台数を意識したコードを生成する場合には、非常に結び付きの強い（つまり粒度の細かい）複数の処理列を生成することがある。これらの処理列に対して、システムの負荷が重いからといって異なった時刻に個別に実プロセッサを割り付けても、同期のためにビジーウェイトでプロセッサを浪費するか、OS を介しての同期でコンテクストを切り換えが頻繁に起こりオーバーヘッドがかさむだけである。そこで、粒度はまさに並列処理が要求している実プロセッサ間の結合の強さ（粒度が細かいほど強い）と考えて、マルチプロセッサ上でグループ単位で実プロセッサを管理する手法とその下でのスケジューリング法について議論する。

2 基本方針と前提条件

基本的に、個別のプロセッサではなく、マルチプロセッサとしてプロセッサの集合体をスケジューリングの単位として利用できることを目的とする。つまり、粒度の細かい処理列はお互いに関連する者同士でグループを形成し、処理列の数に見合う数の実プロセッサの割り当てを要求できるようにする。この要求が満たされる場合は、グループ内の処理列間ではコンテクスト切り換え等のオーバーヘッドなしに同期や通信が実現できる。しかし、指定台数のプロセッサを割り当てるという方針を常に守ることは得策とはいえない (図1参照)。

一つのグループに指定台数を割り当てた後に、実プロセッサが余っており、その台数が実行待ちのグループにとっては不十分である場合、ハードウェア的に台数が揃わないと実行できないグループでない限り、少々オーバーヘッドを覚悟しても実行した方が得策である。(この実行には問題点があり、必ずしも得策とならない可能性がある。これについては6章で言及する。) 本稿ではスループットを落さずに、できる限り要求された指定台数を守るスケジューリング方式について議論する。

システム上のアプリケーションとしては、処理量が多く並列処理の導入が必須である物を仮定している。そして、実時間の応答を要求するような入出力デバイスの割り込み処理等は専用のプロセッサまたはコントローラが処理し、マルチプロセッサは扱う必要がないものとする。ただ、マルチ・ユーザは考慮に入れて、負荷が一時的に大きい時にでもユーザへの反応が極端に遅れることのないように、タイム・スライスでプリエンプションおよび OS による再スケジューリングが実行される。ここで、タイム・スライスは 100ms ぐらいのオーダーを想定している。

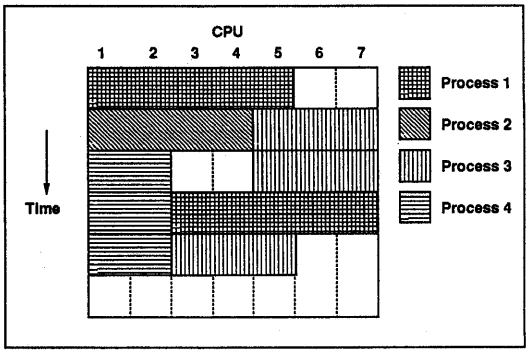

図 1: プロセスの箱詰め問題

スケジューラとしては OS とユーザレベルの 2 階層のスケジューリング方式 [4] をとっている。OS はグループに対して、複数台数の実プロセッサを同時に割り当てる。ユーザ側のスケジューラ (ユーザレベルスケジューラ) は一度割り付けられたプロッサをそのグループ内の処理列に最も適した方法で割り付けることができる。また、指定台数に満たない実プロセッサしか割り当てられなかった場合、頻繁にユーザレベルスケジューラがコンテクスト切り換えを行なう必要が発生するが、カーネルによりコンテクスト切り換えを行なう場合に比べれば、ユーザ／カーネルのモードの切り換えとそれに付随する処理が必要無い分、オーバーヘッドを少なくすることができる [5]。

複数のプロセッサが関連するハードウェアのコンテクストの退避や再設定はプロセッサ・ローカルには行えないので、これらの資源の管理は OS 側で行う必要がある。

3 プロセス、スレッド、シュレッド、ジョブ

本稿で扱う概念の定義を行なう。

3.1 プロセス

関連する一固まりの仕事をプロセスという。プロセスは I/O 要求の単位である。Mach [6] のタスクと同様に、プロセスは 1 つのアドレス空間をもち、その中には複数の実行コンテクスト（スレッド）が存在する。但し、同時には以下に述べる 4 種類のスレッドの内の 1 種類のみが存在する。プロセスは各々独立したユーザレベルスケジューラを持つ。

3.2 スレッド

一台のプロセッサに割り当てられるプログラムの実行の軌跡のことをスレッドと呼ぶ。1つのプロセス内のすべてのスレッドは同一のアドレス空間を共有する。スレッドにはさまざまな粒度のものが考えられる。その粒度の差に注目したマルチプロセッサ上の実プロセッサの管理とスケジューリングを提案することが本稿の目的である。スケジューリング時に利用する情報として一般化したものとして、粒度の大きさやスレッド間の結合の度合を数値化したものが考えられるが、それを完全に扱おうとするとスケジューリングのための情報が膨大になり処理が重くなる。また、静的解析でこれらの数値を求めることはほとんど不可能であるし、実行時に測定したとしても現在の状況を正確に反映したものとはなり得ない。

そこで、静的に決めることのできる粒度の差に注目して、スレッドを以下の4種類に分類し、その情報をスレッドの属性として持たせることにする。

シュレッド 並列化コンパイラなどによって、コンパイル時に特定のプロセッサ台数を仮定して生成したコードを実行するスレッドのことをいう。同じグループのシュレッドは同時に実プロセッサに割り当てられ協調動作を行うことが前提とされ、固定台数であることを利用した最適化がなされている。さらに、複数のプロセッサに関連するハードウェア機構の利用の有無によって、ハードシュレッドとソフトシュレッドに分類される。

- **ハードシュレッド** 「一般化されたバリア型同期機構 (Elastic Barrier)」[7] や「スヌープ・キャッシュ制御機構」[8] などのようなプロセッサ間の同期や通信のハードウェア支援機構をユーザ (コンパイラ) が実行前の静的な解析によって直接用いる場合、必ずコンパイル時に仮定した台数のプロセッサが割り当てられないと動作することができない。粒度としては4種類の中でもっとも細かい。これを特にハードシュレッドと呼ぶ。
- **ソフトシュレッド** ハードウェアの並列実行支援機構を用いずに、共有変数を用いてソフトウェア的に通信や同期を行ないながら処理を進めていくもの。原則的には仮定した数のプロセッサが必要である。粒度はハードシュレッドよりは少し粗い。

スレッド数可変コード 複数のスレッドで密に結合した並列協調処理を行っているが、実行コード自身が実行時の実プロセッサの割り当て台数の変化に対応できるようになっているもの。例えば、DO ループの場合カウンタ変数を排他的にアクセスしさえすれば全てのイテレーションをスレッドで取り合って並列実行できる。したがって割り付けられたプロセッサの数が実行時に増減したとしてもそれにつれてスレッドの数を増減すれば無駄な待ちやコンテクスト切り換えを無くすことができる。粒度はソフトシュレッドと同程度。

単独スレッド 粒度がある程度大きいもので、単独で処理を進め、必要な時にセマフォなどで同期を取り合う。つまり、コンテクスト切り換えやシステム呼び出しのオーバーヘッドが粒度に比べ十分小さいもの。システムの負荷が重い時には、並列処理ではなく時分割の平行処理または逐次処理でも良いもの。通常、生成されてから消滅するまで、もしくはメッセージやメッソドで起動させられてから停止するまで一度も他のスレッドと同期を取らずに実行できる。Mach の cthread [9] などに相当する。

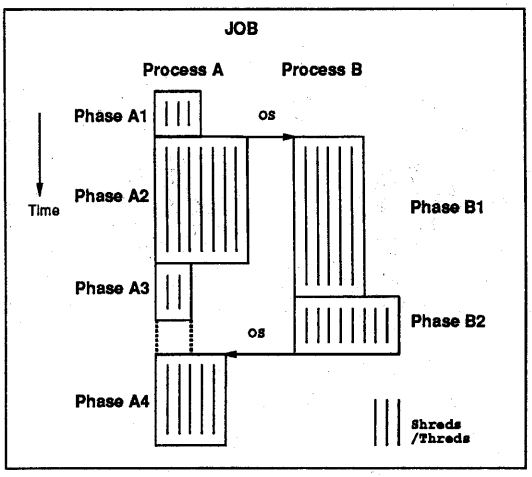

図 2: ジョブ概念図

このように一つのまとまった処理 (プロセス) を実行する場合、それに対するプロセッサ割り当ての戦略は異なったものにする必要がある。すなわち、ハードウェアの支援を利用する時のように必ず決まった台数のプロセッサを必要とするもの、決まった台数のプロセッサを必要とするが緊急時には多少プロセッサの台数が減ってもいいもの、任意の台数のプロセッサで実行できるものの3種に分類できる。それらに対しては異なったプロセッサの割り当て方をしてやる必要がある。

プロセスの定義において同時には1種類のスレッドしか存在できないことになっている。一般化としては、プロセス内に同時に複数種類のスレッドの存在を許すことも考えられるが、これらのスレッドの生成過程を考慮すると、混在するようなケースは考えにくい。このため、不必要な一般化で処理を複雑にするだけと判断し、1種類に限定している。

なお、ハードシュレッド以外のスレッドでは、共有メモリ上の同期変数を使って同期を行う。同期待ちの状態になった時は同期の成立までただビジーウェイトするのではなく、プロセス内のスレッドへの実プロセッサ割り付け状態の監視をスピン・ループ内に含むビジー・ウェイト (SS-Wait: Snoopy Spin Wait) [4] を用いる。監視の結果、無駄なスピンだと判断されるときは自分のプロセスのユーザレベルスケジューラに実プロセッサの制御権を譲り、プロセス内の他のスレッドを起動してもらう。これにより、実プロセッサに割り付けられていないスレッドが処理を行うのを待って、プロセッサ資源を浪費することが避けられる。

3.3 ジョブ

協調して処理を行なうプロセスの集まりをジョブという (図2)。同じジョブ内のプロセス間では共有メモリを持つことができ、プロセス間の同期操作は OS を介して行なう。協調処理を行なうことから fair share なスケジューリングを行なうためのヒントとする予定である。

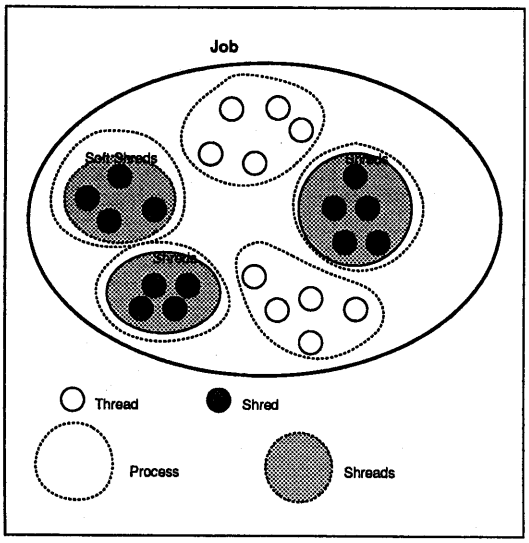

図 3: プロセス、スレッド、シュレッド、ジョブの関係

プロセス、スレッド、シュレッド、ジョブの関係を図3 に示す。

4 プロセスのフェーズ（要求台数, モード）

単体プロセッサで時分割で並行処理を行う場合は、実行コードに粒度やグループに関する情報を必要としない。せいぜい、UNIX の親プロセスがグループとして流用できるぐらいである。(本来の目的は結合の強さを表わしているわけではなく、プロセスの生成・消滅の管理のための情報として用いている。) 従来のマルチプロセッサ用の OS は単体プロセッサ上での OS の時分割の並行処理を並列処理に置き換えただけのものが多かった [10, 11] 。このため、粒度を考慮してグループ単位での実プロセッサの管理のためには、グループに関する新しい情報を OS のスケジューリングに導入する必要がある。

プロセス内には、粒度および実行形態によって 4 種類に分類したスレッドの内の 1 種類しか一時点において存在しない。つまり、プロセスが関連するスレッドの粒度と実行形態別のグループを表わしていることに他ならない。よって、実プロセッサをプロセス単位に割り当てればよいことになる。OS がスケジューリング時に利用する情報として、プロセスに対してフェーズという属性を設ける。フェーズにはプロセスが希望するプロセッサの台数（要求台数）と、その台数をどの程度厳密に OS が守る必要があるかを示す情報（モード）が含まれる。フェーズが持つモードには以下の 3 種類がある。

strict mode 要求台数のプロセッサが必ず必要。足りない場合はプロセッサを全く割り付けない。

moderate mode 要求台数のプロセッサが必要。それ以下の場合多少効率は落ちるが実行可能。

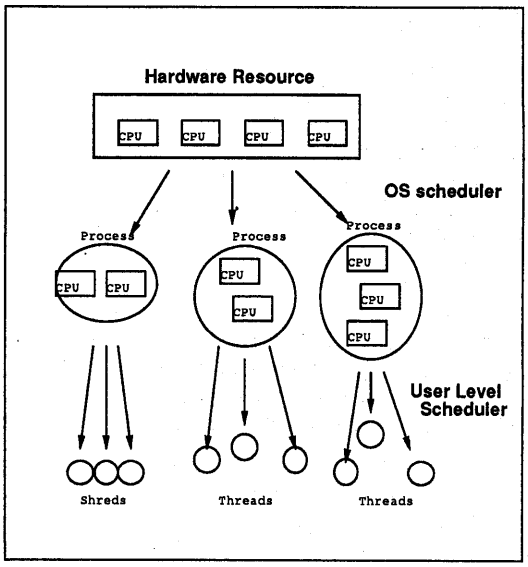

図 4: 2 階層のスケジューリング

loose mode 要求台数かもしくはそれ未満の任意の数のプロセッサを割り付けても処理が可能。

OS のカーネルレベルスケジューラは、システムの負荷とフェーズに従ってプロセスに対して要求台数またはそれ未満の実プロセッサを割り当てる。プロセスが受け取った実プロセッサはそのプロセスのユーザレベルスケジューラによって、スレッドに割り当てられる (図4参照) 。

プロセスはフェーズ（要求台数とモード）を指定してプロセスを生成する。プロセスは実行時に動的にフェーズを変更することができる (図2参照) 。但し、フェーズの変更はカーネルへのシステム呼び出しとなるため、頻繁に行うとオーバーヘッドになる。

他のモードから strict mode に移行した場合や、strict mode 内で要求台数が増やされた場合は、プロセッサ台数の要求が完全に満たされるまでそのプロセスの実行はブロックされる。それにより、解放されたプロセッサはカーネルレベルスケジューラにより他のプロセスに割り当てられる。

5 OS のスケジューリング戦略

プロセスへプロセッサを割り付ける際にはプロセスの持つプライオリティとフェーズ（要求台数、モード）をスケジューリングのパラメータとする。スケジュールの戦略として以下のような方針を採用する。

1. プライオリティの高い順に要求台数のプロセッサが余っているかどうかを調べていく。もし余っているならば要求された数のプロセッサを割り付ける。
2. プロセッサが余っていて、loose mode のプロセスがあればそれにプロセッサを割り付ける。
3. それでもまだプロセッサが余っていて moderate mode のプロセスがあったならそれにプロセッサを割り付ける。

プライオリティはプロセスに割り当てられたプロセッサ数と走行時間の積を用いてエージングを行なう。moderate mode で要求台数のプロセッサが割り当てられなかった場合にはあまりプライオリティを下げないような配慮を行なう。

6 ユーザレベルスケジューラ

OS はプロセスにプロセッサを割り付けるだけで、プロセス内のスレッドとプロセッサとの対応については関知しない。したがって、カーネルによってプロセスに割り当てられたプロセッサをどのようにスレッドに割り付けるかはユーザレベルのスケジューラよらなければならない。

前述のいくつかの実行形態は次のようにして実現される。

ハードシュレッド プロセスのフェーズを strict mode に設定し、コンパイル時に仮定したシュレッド数と同じ数のプロセッサを要求する。ハードシュレッドとして実行中はユーザレベルスケジューラが関与する余地はない。

ソフトシュレッド プロセスのフェーズを moderate mode に設定し、コンパイル時に仮定したシュレッドと同じ数のプロセッサをプロセッサを要求する。ハードシュレッドと違いプリエンプションやシステムの負荷状況によっては要求プロセッサ台数以外での実行が起こりうるので、プロセス内で軽くかつ適切にシュレッドを切り換えるためにユーザレベルスケジューラを必要とする。

スレッド数可変コード、単独スレッド フェーズが loose mode であるプロセスで実現する。スレッド数可変コードについては、カーネルによるプリエンプションが何時起きても良いことを保証するためにユーザレベルスケジューラを必要とする。単独スレッドについてはスレッド間で同期を取り合う頻度が少ないので、ユーザレベルスケジューラの必要性は少ないが、スレッドの消滅や寝た時のコンテクスト切り換えや同期のオーバーヘッドを少しでも削減するために、ジョブ内の単独スレッドをまとめて一つのプロセスとしてユーザレベルスケジューラを設ける。

ここで、2章で触れた余りのプロセッサを要求台数に満たないプロセス (特にソフトシュレッド) に割り当てる場合の問題点について述べる。この場合、ユーザレベルスケジューラを介して頻繁にコンテクスト切り換えが発生すると考えられる。キャッシュ等を用いてデータ通信の量を削減しようとしていても、スレッドが割り当てられるプロセッサが頻繁に変化し、データ通信路に大きなトラフィックを発生させる可能性がある。プロセッサ間のデータ通信路が完全結合でない限り、このことがデータ通信路の競合を引き起こし、他のプロセスの実行に対してレイテンシを持ち込む可能性を増大させてしまう。ソフトシュレッドの粒度の大きさをある程度コンパイラが見積もれる場合は、その情報を使って余りのプロセッサに割り当てるかどうかの判断を行う。

7 プリエンプション

moderate および loose モードのプロセスではユーザレベルスケジューラはプロセス内のスレッド／シュレッドの状態を管理するキューを持ち、実行可能状態にあるスレッドを選び出しては、それに制御を移す。スレッドが SS-Wait で飢餓状態を検出した場合や、陽にセマフォなどの同期操作を行なった場合は、自らユーザレベルスケジューラに制御を移す。

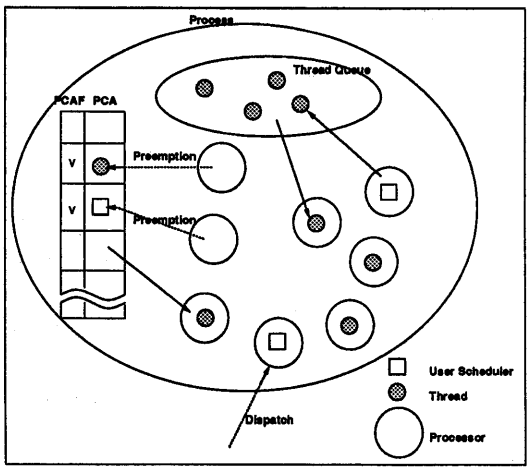

図 5: スケジューラとプリエンプションの関係

moderate および loose モードのプロセスでは個別のプロセッサごとにプリエンプションが起こる。執行猶予期間を持った時限爆弾式のプリエンプション方式であれば、その執行猶予期間内にスレッドを切りの良い所まで実行して自ら実行権を放棄すれば良い。しかし、この方法はユーザ・プログラム (実行コード) に余分な制約を加える。

任意の時点で予告なくカーネルがプロセッサのプリエンプションが可能で、かつプロセス内のすべてのスレッドがプリエンプトされたスレッドのために飢餓状態になるのを避ける方法があれば、その方が望ましい。この条件を満たす一方法を以下に示す。

プリエンプションの時点で走行中であったスレッド（シュレッド）のコンテクストがカーネルのデータ構造中に格納されてしまうと、ユーザレベルからはそのコンテクストに触れなくなってしまうのでユーザレベルスケジューラでは再スケジューリングができない。そこで、カーネルがプロセッサをプリエンプトする時にはプリエンプトされたスレッドのコンテクストを必ずユーザレベルから参照のできる領域にセーブする。この領域 (PCA: Preempted Context Area) はプロセスごとに実プロセッサの台数分だけ確保される。個々の PCA に対しては、PCA 中に有効なコンテクストが格納されているかどうかを示すフラグビット (PCAF: PCA Flag) が用意されている (図 5)。PCAF はユーザ／カーネルから読み書き可能である。カーネルがあるプロセッサをプリエンプトする時には

1. まずその上で動いているスレッド／シュレッドの動作を止め、
2. そのコンテクストを対応する PCA に格納し、
3. 対応する PCAF を立てる。

PCA にはすでにコンテクストが格納されていることも起こり得るが、その時は単に新しいコンテクストを捨てることで解決される。

カーネルがプロセッサをプロセスに割り付ける時は必ず

ユーザレベルスケジューラに制御を移す。この場合や、同期等でユーザスケジューラが起動された場合、ユーザレベルスケジューラは、

1. PCAF をスキャンしてコンテクストが格納されている PCA を捜す。
2. 見つかった場合は PCA からスレッドのコンテクストを読みだして PCAF を落し、そのスレッドに制御を移す。
3. 見つからなかった場合は通常の処理として、実行可能なスレッドをキューから選び出して制御を移す。

ただし、ユーザレベルスケジューラ間で PCA, PCAF のアクセスの競合を排他制御する必要があり、スレッドのキューの操作も排他制御が必要である。そこで、ユーザスケジューラ間で 1 つのグローバルロックを設けて排他制御を行なうことにする。同時に 2 つ以上ののユーザレベルスケジューラが PCA をアクセスすることはない。

このようにすると、グローバルロックを取ったスケジューラがプリエンプトされた時が問題である。しかし、グローバルロックを SS-Wait によるロック操作で行なうことにすれば、ロックを取ったているスケジューラがプリエンプトされたことを検出することができる。その時には、ロック待ちのスケジューラ間で選抜を行なう。勝ち残ったスケジューラは自分のスケジューラコンテクストを捨て、PCA 中に格納されているロックのオーナのコンテクストを実行する。

8 おわりに

マルチプロセッサ上で粒度が細かく密に協調並列動作するプログラムの存在を考慮に入れたプロセッサ資源の管理法およびスケジューリングの戦略について述べた。実行コードの生成過程から静的に判断される粒度の情報を利用して、並列動作するプログラムを 4 種類のカテゴリーに分類した。この分類を利用した管理ならびにスケジューリングを行うため、実プロセッサの割り当て単位であるプロセスに要求台数とモードという属性を付加した。さらに、ユーザとカーネルの 2 階層のスケジューリング方式で使用できるカーネルによるプリエンプション方式を提案した。この方式はプリエンプションの予告を伴わず、従来の時限爆弾方式よりも優れている。今後はこの方式を用いたプロトタイプの作成や、シミュレーションなどによる評価を行なう予定である。

今回は I/O 処理などのシステムコールやページトラップの制御については何も述べなかったが、それらも当然スケジューリングのヒントとして導入していきたい。

また今後の課題として、NUMA(Non-Uniform Memory Access) のマルチプロセッサ上でのスケジューリング、動的なプログラムとデータの配置および移送について、実行前に静的に得られる情報がどう生かせるか検討していきたい。

9 謝辞

本研究の機会を与えて下さった IBM 東京基礎研究所の山内 長承氏、上村 務氏と鈴木 則久所長に感謝いたします。また、筆者の一人である松本を研究生として御指導いただいている東京大学工学部の田中 英彦教授とお世話になっている田中研究室のみなさんに感謝いたします。

参考文献

[1] Allen, F. et el.: *An Overview of the PTRAN Analysis System for Multiprocessing*, Journal of Parallel and Distributed Computing 5, Academic Press, pp. 617-640, August 1988

[2] 本田弘樹 他: "OSCAR 上での Fortran 並列処理系のインプリメントと性能評価", 信学技報 Vol. 89 No. 168 CPSY 89-57, pp. 75-80, August 1989

[3] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.*, In Wada, E.(Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, June 1986, pp.209-216.

[4] 松本 尚: "マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察", 計算機アーキテクチャ研究会報告 No.79-1, 情報処理学会, November 1989, pp.1-8.

[5] 渦原 茂, 森山 孝男: "マルチプロセッサシステムにおけるライトウエイトプロセス機構", 日本ソフトウエア科学会第 6 回大会, October 1989

[6] Robert V.Baron, David Black, et.al. (Carnegie Mellon University): *MACH Kernel Interface Manual*, April 12 1990

[7] 松本 尚: "一般化されたバリア型同期機構の諸問題について", 並列処理シンポジウム JSPP '90 論文集, May 1990, pp.49-56

[8] 松本 尚: "細粒度並列実行支援機構", 計算機アーキテクチャ研究会報告 No.77-12, 情報処理学会, July 1989, pp.91-98.

[9] Eric C.Cooper, Richard P.Draves (Carnegie Mellon University): *C Threads Manual*, July 1987

[10] 穂積 他: "TOP-1 オペレーティングシステム", 第 39 回情報処理学会全国大会, 1989

[11] *Symmetry Technical Summary*, Sequent Computer Systems, inc.