計算機アーキテクチャ　82－3
（１９９０．４．２０）

可変構造型並列計算機の並列オペレーティング・システム
－プロセス管理とメモリ管理－

恒冨邦彦†，草野和寛†，福澤祐二‡，福田 晃†，村上和彰†，富田眞治†
（†九州大学大学院総合理工学研究科，‡ソニー（株））

「可変構造型並列計算機（Reconfigurable Parallel Processor）」は，我々が現在開発中の汎用／多目的マルチプロセッサ・システムであり，メモリ／ネットワーク・アーキテクチャを種々の並列処理形態に適応させることが可能である．このシステム上に現在構築しているオペレーティング・システムは，可変構造型並列計算機を密結合型マルチプロセッサとして用いており，カーネル内部をメッセージ指向により構築することにより，データパラレルな処理を実現し，カーネルの負荷を分散する．また，密結合型マルチプロセッサでの並列処理モデルとしてユーザに対し軽いプロセス（スレッド）の機能を提供する．プロセッサの割当て，解放をプロセス単位に行うスケジューリングを用いることにより，スレッド切り換えのオーバヘッドを軽減できる．また，スレッド間の同期機構として，システムの負荷に応じてサスペンドロックに移行するタイムアウト付きのスピンロックを用いることによって，コスケジューリングより柔軟なスケジューリングを実現する．メモリ管理については，アドレス変換テーブルの管理とスタックの割当て方法について述べる．アドレス変換テーブルをコピーする方式としない方式を評価し，後者を選択する．また，スタックへの高速なアクセスを実現するために，スタックの遅延生成を提案する．

A Parallel Operating System for The Reconfigurable Parallel Processor
－Process Management and Memory Management－

Kunihiko TUNEDOMI†, Kazuhiro KUSANO†, Yuji FUKUZAWA‡, Akira FUKUDA†,
Kazuaki MURAKAMI† and Shinji TOMITA†

† Interdisciplinary Graduate School of Engineering Sciences, Kyushu University,
6-1 Kasuga-koen, Kasuga-shi, Fukuoka, 816 JAPAN
tunedomi@kyu-is. is. kyushu-u. ac. jp

‡ SONY Corporation

A reconfigurable parallel processor under development at Kyushu University is a MIMD-type multipurpose multiprocessor system. This system can be tailored to a broad range of applications. Its operating system under development considers it as a tightly coupled multiprocessor. Its kernel is based on the message oriented method to implement data-parallel processing. The operating system provides users with functions of the light weight process (thread) as a parallel processing model for tightly coupled multiprocessor system. The overhead of the thread switching can be reduced by our scheduling scheme where processors are allocated and deallocated to each process. The spin lock mechanism with time-out facility is employed to realize more flexible sheduling than coscheduling. We evalute two methods where address mapping table is copied and it is not copied. As a result, the latter is employed. We propose a method of lazy ceation of stack to have fast access to a stack.

1. はじめに

我々が現在開発中の「可変構造型並列計算機(Reconfigurable Parallel Processor)」[1][2]は，汎用／多目的マルチプロセッサ・システムであり，128台のプロセッシング・エレメント（PE）が128×128のクロスバー網により接続されている．本システムは，種々の並列処理形態に対して計算機構成を適応させることを目的としており，相互結合網およびメモリにダイナミック・アーキテクチャを採用している．これによりシステムは図1に示すアドレス空間を持ち，密結合型／疎結合型マルチプロセッサ双方をハードウェア・レベルで実現できる．このシステム上に構築するオペレーティング・システム（OS）[3][4]は，アドレス空間を制御管理するメモリ管理と，プロセス管理が必要となる．本稿では，このOSのプロセス管理とメモリ管理について述べる．

2. オペレーティング・システムの設計方針

本システム開発の主な目的を以下に示す．

（1）ハードウェア・アーキテクチャと各種並列処理問題との親和性の検討

（2）解くべき並列処理問題にハードウェアを動的に適合させることによるマルチプロセッサ・システムの応用分野の拡大

（1），（2）を遂行するためには，並列／分散OSの研究を含めた広範囲な並列処理の研究が不可欠である．（2）を実現するためには，まず（1）の検討すなわち，ハードウェア・アーキテクチャと各種並列処理問題との親和性を定量的に検討する必要があると考える．そこで，我々は（2）を実現するOSの開発に先立って，（1）を対象としたOSを開発する．

現在開発中の，密結合型メモリ・モデルを基本としたOSの設計方針を以下に示す．

（1）密結合方式に適した並列処理モデルをユーザに提供する．

ユーザプログラムをマルチプロセッサ上で中粒度の並列処理（スレッド）に分割して並列実行させる機能を提供する．詳しくは3章で述べる．

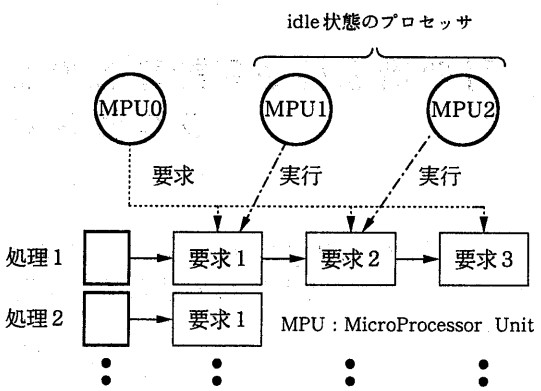


図2 カーネルのデータパラレル化

（2）カーネル内部を可能な限りデータパラレル化する（負荷分散）．

ある1つのシステムコールの実行軌跡である制御フローに着眼した場合，従来の密結合型マルチプロセッサ用OSでは以下の方式が採られている．

（a）制御フローが1つ，あるいは複数のプロセッサで実行されるにしても，いずれの場合でも1つの逐次的な制御フローとして実行される．

（b）（a）の方式をさらに進めて1つのシステムコールを機能分割し，並列実行できるところは並列実行する（コントロールパラレル化）．

我々は，これらの方式をさらに進めて複数のスレッドの同時生成などデータパラレル化できるところはデータパラレル化する．すなわち，オーバヘッドの問題に対処する必要があるが，密結合型マルチプロセッサ用OSの内部処理のデータパラレル化の可能性を追求する．

これを実現する方式として，メッセージ指向のOSを構築する．すなわち，図2に示すようにデータごとの処理に各機能を細分化し，実行要求をメッセージとして各処理のキューにつなぐメッセージ方式により，カーネルの内部処理を並列化する．

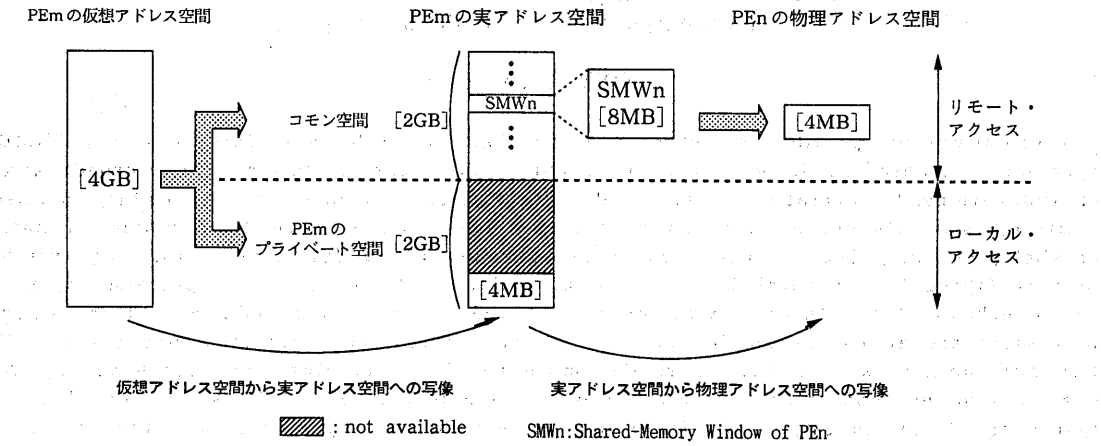


図1 可変構造型並列計算機のアドレス空間写像

(3) 多様なシステムコールを提供する.

ユーザが多様な並列処理問題を扱えるようにするため，多様なシステムコールを提供する．また，ソフトウェア資産の継承という観点から，一般的なOSであるUNIXと互換性をもたせる．

3. プロセス管理

3. 1 並列処理モデル

UNIXをはじめとする従来のシングルプロセッサ上のOSでは，実行の単位としてプロセスを考え，同時に複数のプロセスが平行に実行できる環境を提供し，またスループットを向上するために時分割スケジューリングを行っている．従ってマルチプロセッサ上では，これらのプロセスの並列実行が可能となり，各プロセスがお互いに独立である場合には処理が高速になる．しかし，ある仕事内が並列に実行可能であり，互いに協調動作を行う処理が存在する場合に，これをプロセスに分割して実行させると通信や同期のオーバヘッドが大きくなるなどの問題が存在し，プロセスモデルによる記述は並列処理には適しているとはいえない．

我々は，ユーザのプロセス内部を並列実行可能な処理の流れ（スレッド）に分割して実行することによりこの問題を解決する．同一プロセス内の複数のスレッドは仮想アドレス空間を共有しており，通信や同期処理を1つのユーザ空間のみで処理することが可能となる．プロセスは，1つの仮想アドレス空間とその空間に割り当てられた資源および複数のスレッドからなる．スレッドは，プログラム・カウンタ，レジスタ，スタック領域および実行に関する情報からなる．

3. 2 スレッドに関するシステムコール

スレッド操作の主なシステムコールを，表1に示す．プロセスが新しく生成されるときには，スレッドは1つ存在する．プロセス内の処理が並列実行可能である箇所においてユーザがシステムコールthread__lfork（n）またはthread__sfork（n,func,arg1,・・・）を実行させることによって複数（n個）のスレッドを生成し，処理を並列に行うことが可能となる．このときシステムコールの戻り値としてスレッドidが各スレッドに返される．スレッドidは0番から昇順にプロセス内でユニークに割り当てられるので，これを用いて配列データを各スレッドに割り振ることができる．並列実行が終了しスレッドを1つに収束するときは，スレッド終了のシステムコールthread__join（）を実行する．スレッドidが最も小さなスレッドを除く全てのスレッドがこのシステムコールにより消滅され，プロセス内のスレッドは1つになる．

表1 スレッドの主なシステムコール

関数名	関数引数	戻り値	機能
thread__lfork	n：スレッド数	スレッドid or エラー	スレッドをn個生成し，次の命令から実行する
thread__sfork	n：スレッド数 関数，関数引数	スレッドid or エラー	スレッドをn個生成し，funcを実行する
thread__join	なし	成功 or エラー	複数のスレッドを1つに収束する
thread__suspend	スレッドid	成功 or エラー	スレッドを中断する
thread__resume	スレッドid	成功 or エラー	スレッドを励起する

3. 3 メッセージ指向によるプロセス管理構築

各スレッドにはスレッドコントロールブロック（THCB）とスタック領域が割り当てられ，実行時の情報などをTHCBに格納している．従って，thread__lfork（n）によって一度に多量のスレッドを生成すると，THCBも多量に必要となり，これを初期化する処理が逐次的になされる限りボトルネックとなる．

この問題を解決するために，設計方針で述べた様にメッセージ方式によりプロセス管理を構築し，処理を並列化する（図3）．スレッド生成のシステムコールを実行しTHCBの初期設定を行うときには，THCBの初期化処理サーバのキューに，生成されるスレッドごとに要求をつなぐ．idle状態のプロセッサはキュー検索を行うループを実行しており，キューから要求を1つ取り出し実行する．複数のidle状態のプロセッサが存在すれば，各々のスレッドのTHCBが並列に初期化される．このように複数スレッド生成の並列化を実現できるが，このとき，メッセージ操作のオーバヘッドが問題となる．すなわち，並列化とそれに伴うオーバヘッドのトレードオフの問題である．これに関しては，複数スレッド生成のシステムコールを受け付けたプロセッサでまとまった数のスレッドを生成し，残りのスレッド生成の依頼をメッセージにして他のプロセスで処理をする方法などが考えられる．

3. 4 スケジューリング

3. 4. 1 スケジューリングの要件

マルチプロセッサ上のプロセス管理で重要なことは，プロセッサのスケジューリングである．スケジューリングによってシステム全体のスループットが左右され，また通信や同期のためのオーバヘッドを極力回避することができるからである．スケジューリングには，仕事のプロセッサへの時間的，空間的マッピングを実行以前に行う静的スケジューリングと，システムの状況により実行時に行う動的スケジューリングがある．我々は両方について研究しているが，ここでは，動的スケジューリン

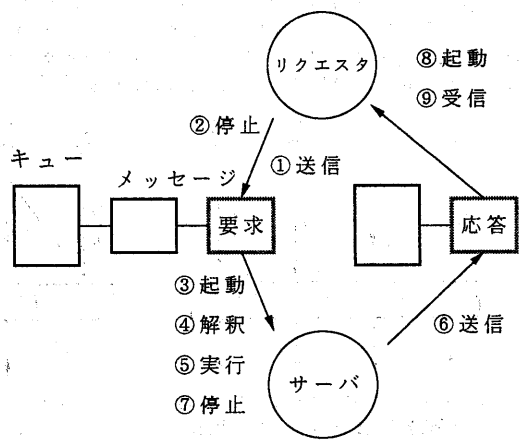


図3 メッセージの処理

グについて述べる．通信パターン，通信コストを考慮した静的スケジューリングについては文献[5]を参照されたい．

今N個のプロセッサが結合されているマルチプロセッサを考える．UNIXのマルチプロセスの時分割スケジューリングをそのままマルチプロセッサに適用すると，実行可能なプロセスのキュー（プロセスキュー）を用意し，キューを検索して優先度の高いものからN個選んで各プロセッサに割り付け，プロセッサはそれぞれの優先度に対し割り当てられる実行時間（タイムスライス）だけ処理を行うこととなる．タイムスライスを使いきると，プロセスの優先度を再計算し，最高の優先度を持つ実行可能なプロセスを選択して再びユーザプロセスを実行する．

スレッドは実行の単位であるから，上記のスケジューリングにおいてプロセスをスレッドに置き換えてそのまま適用すると，実行可能なスレッドのキュー（スレッドキュー）がシステム全体で共有され，そのキューの優先度の高いスレッドがN個実行されるスケジューリングとなる．しかし，スレッドが同一プロセス内処理の並列性の記述のために用いられ，複数のスレッドが特に密接な関係にあることから，上記の方式は次の問題が生じる．

（1）同期によるオーバヘッドの増大

同一プロセス内のスレッド間の同期としては，ユーザ空間において共有のロック変数を持ち，これがアンロックされるまで繰り返しreadするスピンロックが有効である．これによりスレッド間の同期がカーネルの処理を必要としないために高速に行える可能性がある．しかし，同一プロセス内の複数のスレッドが生産者と消費者の関係にある場合には，両者が共に実行されていることが保証されていなければ能率が上がらない．生産者があるデータの生産を行う以前に消費者が励起されて実行を開始しても，データへの排他制御のためのスピンロックを繰り返すのみでタイムスライスを消費し，資源の無駄使いとなる．上記のスケジューリングでは，同一プロセス内のスレッドがばらばらに実行されることになるので，この問題が生じる．

（2）並列性を実現する上でのオーバヘッドの増大

上記のスケジューリングでは，スレッドのタイムアウト毎にスレッドの優先度を各プロセッサが再計算することになるので，システム内に存在するプロセッサの台数より遥かに大きい並列性をスレッドを用いて実行する場合，スレッドの切り換え回数が増大する．

（3）スレッド切り換えによるオーバヘッドの増大

各プロセッサのキャッシュ上には，そのスレッドが用いた情報がキャッシングされている．従って，過去に実行されたスレッドが再び励起される場合およびスレッドを新たに実行する場合には，そのスレッドと同一プロセス内のスレッドの処理を実行していたプロセッサに割り当てられると，キャッシュ内のデータの再利用が可能となり処理が高速となる．上記の方式では，プロセス単位のプロセッサの割当てを考慮していないので，新たに実行されるスレッドが異なるプロセス内のスレッドとなる可能性が大である．このとき，キャッシュのミスヒットが頻発すると共に仮想空間の切り換えのオーバヘッドが生ずる．

以上述べた問題点（1）～（3）を解決するスケジューリングを考える．

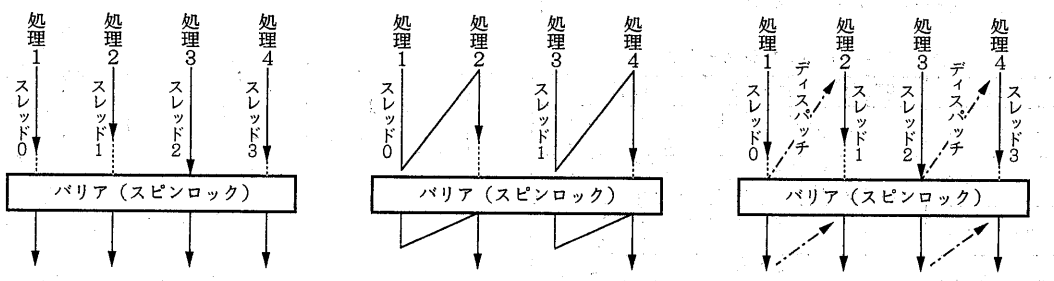
3. 4. 2 コスケジューリング

上記の問題を解決するためにMedusa [6] をはじめ多くの並列OSがコスケジューリングを行っている．コスケジューリングとは，プロセスキューにプロセス内に存在するready状態のスレッド数すなわち並列処理数（並列度）をもたせ，このプロセスを実行に移すときは並列度だけプロセッサを割り当てて処理が並列に実行されることを保証する方式である．これによって，（1）の問題は解決できる．しかし，並列度がシステム内のプロセッサ数を越える場合，また割当て可能なプロセッサ数が並列度より少ない場合などは柔軟に対処することができない．

これを回避する方法として，ユーザによって指定された並列度よりidle状態プロセッサ数が少ない場合に，並列度をidle状態のプロセッサ数へ再分割する方法が挙げられる（図4（2））．すなわち，システムコールにより現在のidle状態のプロセッサ数を確認し，idleプロセッサ数だけに並列度をマージし，それらをスレッドとして実行する．しかしこの方式では，一度並列度をidleプロセッサ数分だけにマージさせてしまうと，新しくidle状態のプロセッサが発生した場合でも，並列度を再び上げることができない．

3. 4. 3 本OSにおけるスケジューリング

コスケジューリングは動的に変化するシステムのidle状態のプロセッサ数を反映できない欠点を持つ．これを解決するスケジューリング方式として，本OSでは，2段階スケジューリングを実現する．すなわち，第1段階



（1）プロセッサ数が並列度以上の場合（並列処理を同数のスレッドで実行）　（2）プロセッサ数が並列度以下の場合（並列処理をマージして実行）　（3）プロセッサ数が並列度以下の場合（並列処理を同数のスレッドで実行）

図4 割当られたプロセッサ数に着目した並列処理の実行方法

目のスケジュールは，UNIXと同様にプロセッサの実行時間からプロセス優先度を決定し，優先度の高いプロセスからready状態スレッド数に応じてプロセッサを割り当てる．コスケジューリングと違い，割当て可能なプロセッサ数がスレッド数より少ない場合でもプロセッサを割当てる．第2段階目のスケジュールは，プロセスに割当てられたプロセッサをプロセス内のスレッドに対しFCFS（First Come First Served）方式により割り当てる．これにより高優先度のプロセスは全スレッドが同時に処理を行うが，低優先度のプロセスも，一部のスレッドが処理を進めることができる．この処理は，優先順位を持つプロセスキューをシステム全体で1つ共有し，各プロセスごとにスレッドキューを割り当てることにより実現される．この手順を以下に示す．

1）プロセスのスケジューリング

プロセスを実行していない（idle状態の）プロセッサはプロセスキューを検索し，最高優先度のプロセスを選択する．キューの要素には，プロセスidとそのプロセス内の実行可能状態（ready状態）のスレッド数が記録されており，プロセッサが割り当てられる毎にこのスレッド数を減ずる．プロセスの処理中に新たなスレッド生成が発生する場合（図5（1））およびスレッドがサスペンド状態から励起されて実行待ち状態になる場合には，スレッド数に加算する．スレッド数が0となれば，その要素をキューから外す．以上の処理を行い，プロセスを選択したプロセッサは2）のスレッドのスケジューリングに移る．

2）スレッドのスケジューリング

スレッドキューはプロセス対応に割り付けられる．プロセッサは実行するプロセスのスレッドキューを検索し，キュー先頭のスレッドを実行する．実行中のスレッドがI／O待ちや，ページフォールトの処理などでサスペンドされた場合，再び同一プロセスのスレッドキューからready状態のスレッドを選択し処理する．図5（2）に示すようにready状態のスレッドが存在しない場合にはじめてプロセスから解放され，1）のプロセススケジューリングに戻る．

この方式では，プロセッサのプロセス切り換えは，（a）タイムアウト後に優先度の高いプロセスが発生した場合，（b）割り込みにより優先度の高いプロセスにより横取り（preempt）される場合，および（c）プロセス内のready状態スレッドが存在しない場合に発生する．図6に示すように，最初にプロセスを実行するプロセッサにタイマを設定し，タイムアウトになると自プロセッサはプロセスキューの優先度を再計算する．優先度は，プロセッサの実行時間，初期設定プライオリティ等により決定する．このとき，実行状態プロセスよりも高優先度のready状態プロセスが発生すると，低優先度の実行状態プロセスのプロセッサに割り込みをかけ，再スケジューリングを行わせる．これが（a）（b）の切り換え処理である．このようにタイムスライスの終了判定および優先度の再計算は，プロセスが起動されたときに最初に実行されるスレッドを実行していたプロセッサが行い，各プロセッサが行う必要がない．また，ディスパッチの対象とならないプロセスのスレッド処理はタイマを実行するプロセッサを除いては実行が中断されることがなく，スレッドのプロセッサ間の移動も発生しない．従って問題点（2）が解決できる

2段階のスケジューリングによりプロセッサはプロセスごとに割当てられ，タイムスライス内でスレッドの切り換えが発生した場合，同一プロセスのスレッドを優先して実行させる．これにより，仮想空間の切り換えやキャッシュヒット率の低下を抑えることができ問題点（3）が解決できる．

プロセスの優先度が高いものは，ready状態のスレ

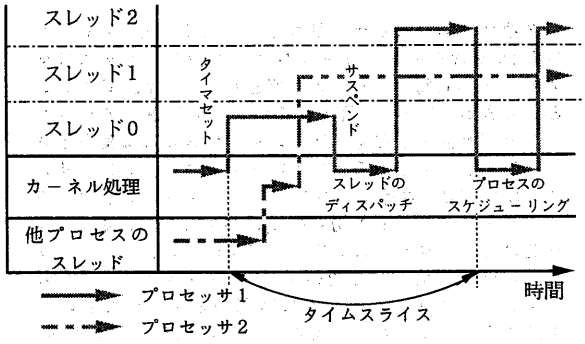
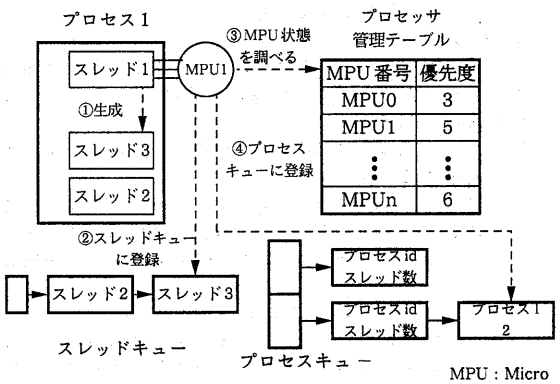
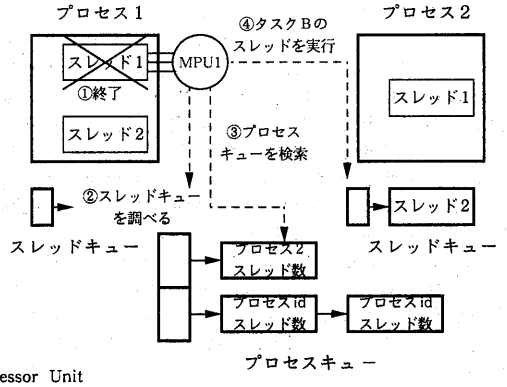


図 6　2段階スケジューリングにおけるプロセッサの処理



（1）スレッドの生成
（優先度の低いプロセスを実行中のMPUが存在しない場合）



（2）スレッドの終了
（MPUの割り当てを待つプロセスが存在する場合）

MPU : Micro Processor Unit

図5 スレッド生成と終了処理

ッド数だけプロセッサを割り当てられるので，スレッド数がidle状態のプロセッサ数よりも少ないときには全スレッドが同時に実行され，問題点（1）は回避することができる．しかし，優先度の低いものは割当てられたプロセッサ数が少なく，全スレッドが同時には実行されないことが考えられる．またスレッドのスケジューリングにおいては，実行中のスレッドがサスペンドまたは終了するまでスレッドの切り換えは発生しない．従って優先度の低いプロセスにおいては，スレッドがスピンロックを実行する場合には問題（1）が発生する．これを解決するためにスピンロックに時間制限をつけたものを用意する．スピンロックをある一定期間繰り返した後，ロックが解放されない場合にはスレッドをサスペンドさせ，スレッドキューから新たにスレッドを選択し実行する（図4（3））．

3. 4. 4 スケジューリングの比較

コスケジューリングと本OSのスケジューリングとの比較を問題点（1）～（3）について行う．

問題点（1）については，コスケジューリングは，いかなるプロセスも，その内部処理を同時実行させることによりに完全に解決している．一方，本OSのスケジューリングは，優先度が高いプロセスのみ内部処理を同時に実行させる．並列度より少ないプロセッサで実行される低優先度のプロセスについては問題点（1）は解決されないので，タイムアウト付きのスピンロックを実現する．これにより，タイムスライスを全てスピンロックに費やす可能性は低くなり，問題点（1）を回避できるが，同期によるオーバヘッドがスピンロックの制限時間によって左右される．制限時間が短い場合には，全スレッドが同時に実行されており，スレッドの切り換えを必要としない時にもサスペンドされるスレッドが発生し，スピンロックの同期の軽さをいかすことができない．制限時間の長い場合は，全スレッドが同時に実行されていない時には，早急にスレッド切り換えが必要であるが，タイムスライスのほとんどをスピンロックに費やしてしまい同期のオーバヘッドが大きくなる．また，コスケジューリングのように同一プロセスの全スレッドの処理進行状況をそろえることは保証されないので，全スレッドが実行されている場合でもスピンロックそのものに費やされる時間も増加する．スピンロックの制限時間の検討は今後の課題である．

次に問題点（2）について考察する．コスケジューリング，本OSのスケジューリングとも優先度はプロセスごとに存在する．いま，実行されていたプロセスの優先度よりも高い優先度のプロセスが発生し，このプロセスにプロセッサを割当てたため実行されていたプロセスのプロセッサ数が減少したとする．この場合，コスケジューリングでは，並列度の少ない別のプロセスを起動しなくてはならないのでキャッシュのデータが生かされない．これに対し，本OSのスケジューリングではプロセッサ数が減っても実行を続けるのでキャッシュのデータを使うことができ，また，スレッドの切換えもコスケジューリングに比べ，回数が減少するので問題点（2）を解決する．

問題点（3）については，両方のスケジューリングとも，プロセスが実行を続ける限り，プロセッサはプロセス間を移動しないため，キャッシュの性能を生かすことができる．しかしコスケジューリングは，上記のようにプロセッサ数が不足するとプロセスはディスパッチされ実行を停止するので，プロセスが実行を続ける可能性が本OSのスケジューリングよりも低い．また，同一プロセス内のスレッドは仮想空間を共有しており，このプロセスのスレッドを実行したプロセッサがキャッシングしたデータを他のスレッドが利用できる可能性がある．従って，本OSのスケジューリングでは，プロセッサ数が並列度より少ないときには，プロセッサが同一プロセス内のスレッドを優先して実行するのでキャッシュの内容を再利用することができる．コスケジューリングでは，このようなキャッシュ内のデータの再利用はできない．

4. メモリ管理

4. 1 概要

本システムでは，メモリ構成のダイナミック・アーキテクチャを実現するために，ローカル／リモート・アーキテクチャを採用している．このために，図1に示したように3つのアドレス空間（仮想アドレス空間，実アドレス空間，物理アドレス空間）を導入して，2レベルのアドレス変換を行う．実アドレス空間は，全PEに共通のコモン空間と各PEの私有領域であるプライベート空間から構成されている．メモリ管理では，これら3つのアドレス空間と，2レベルのアドレス変換に用いられるアドレス変換テーブルの管理を行う必要がある．本システムに特徴的な実アドレス空間の管理において問題となるのは，仮想アドレス空間からコモン空間へのマッピングを無効化するのにオーバヘッドを伴うことである．これに関しては，文献〔7〕を参照されたい．本節ではアドレス変換テーブルとスタックの管理について述べる．

4. 2 アドレス変換テーブル

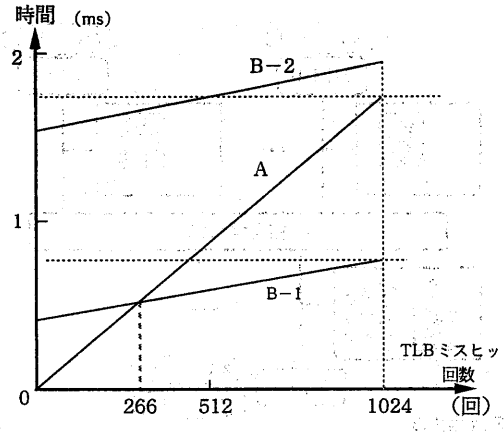
密結合型マルチプロセッサにおいて，複数プロセッサ上で同一プロセスの異なるスレッドが実行されることを考えると，同一プロセス内のスレッドは仮想アドレス空間を共有するので，アドレス変換テーブル（仮想アドレス空間から実アドレス空間への変換テーブル）へのアクセス競合が発生することが考えられる．本システムは，分散メモリ構成のローカル／リモート・アーキテクチャを採用している．このことから，スレッドの使用するアドレス変換テーブルの管理方法について，以下の2つが考えられる．

（1）全てのスレッドが同一のアドレス変換テーブルを使用する．

（2）アクセス競合を避けるために，スレッドが実行されるプロセッサ毎に，または実行されるプロセッサのN個毎にアドレス変換テーブルのコピーを持たせる．

以下にこの2つの方式について考察する．

（1）の方法を用いると，アクセス競合が発生する．しかし，（2）の方法のようにアドレス変換テーブル間のコヒーレンス問題は発生しない．この時に，アドレス変換テーブルと異なるプロセッサ上で実行されているス



A (方式 (1)) ：リモート・メモリへのアクセス
B−1 (方式 (2)) : 1ページのDMA転送
＋ローカルメモリ・アクセス
B−2 (方式 (2)) : SPARCによる1ページの転送
＋ローカルメモリ・アクセス

図7 アドレス変換テーブルへのアクセス時間

レッドはリモート・アクセスを行うことになるが，そのオーバヘッドはハードウェアのTLBやキャッシュにより，かなり軽減されると考えられる．

（2）の方法を用いると，アクセスの競合は少なくなる，またはなくなると言えるが，アドレス変換テーブル間のコヒーレンスを保つことが必要になる．つまり，複数のアドレス変換テーブルの内容が一致していない状態が発生するのを防ぐ必要がある．特に，ページの入れ替え時のコヒーレンスが問題となる．例えば，あるスレッドがページフォールトを起こして，ページをメモリに読み込んで，他のアドレス変換テーブルを変更するまでの間に，それを用いているスレッドが同じページに対してページフォールトを発生させることも考えられる．この場合には，メモリ上に同じページが複数個存在することになり，さらにアドレス変換テーブル間のコヒーレンスが保てなくなる．また，アドレス変換テーブルのコピーの手間もローカル／リモートアーキテクチャをとる本システムにおいてはかなりのオーバヘッドとなる．

以上に述べたように，この問題は，リモート・アクセスと，アドレス変換テーブルのコピー及びコヒーレンス処理との間のトレードオフによりどちらが良いかが決まることになる．

方式（1）（2）におけるアドレス変換テーブルへのアクセス時間を図7に示す．図7から分かるように，TLBのミスヒット回数が数百回（図7では266回）までは，方式（1）の方がアクセス時間が短い．一般にスレッドの大きさはTLBのミスヒット回数が数百回以下となる大きさと考えられること，及び方式（2）におけるコヒーレンス処理のオーバヘッドを考慮すると方式（1）が優れていると言える．以上に述べたことにより，本システムではアドレス変換テーブルは全スレッドで同一のものを使用する．つまり，アドレス変換テーブルのコピーは作らない．

4. 3 スレッドのスタック領域

4. 3. 1 概要

同一プロセスのスレッドは，仮想アドレス空間等の資源を共有しており，各スレッドに固有の情報は比較的少ない．これにより，同一プロセス内のスレッド間でのコンテキストの切り換えを高速に行うことができる．しかしながら，各スレッドで保持しているスタックは同一仮想アドレス空間に存在するので，スレッドのスタックの保護を行う必要がある．これは，各スレッドの複数のスタックが同一仮想アドレス空間上に存在するために生じる問題である．

本システムでは，1つのシステムコールでスレッドを複数作ることを可能にする機能を提供する．これを考慮して，スタックの確保と保護を考えた場合の問題点について検討する．

（1）各スレッドのスタックの大きさをどの程度に設定するか．

本システムにおいてはスレッドの粒度として中程度，つまり，あまり大きくない関数程度に考えている．しかし，使用するスタックの大きさは実行前に知ることが非常に困難である．よって，ある程度大きめに確保しておくことが多い．しかし，メモリの無駄が生じることになる．また，動的にスタック領域を拡張する方式では，スタックを拡張するのにはオーバヘッドを伴わざるをえない．これよりスタックの大きさは，スレッドによるアプリケーションを多数実行して，検討する必要がある．本システムでは，スレッドのスタックを大きめにとり，さらに各スレッドのスタックの大きさを指定可能とする．これは，ユーザまたはコンパイラで予測した値を使えるようにして，メモリを無駄に使用するのを防ぐための機能である．

（2）同時に複数のスタックを高速に確保する方法．

1度に複数個生成するような場合に逐次的に生成した場合と同じ程度の時間がかかるのでは，スレッドの複数生成機能を提供する意味があまりない．そこで，スタックを複数個生成する処理を如何に高速に行うかが問題となる．十分な空き領域が存在する場合には，各スタックの先頭番地を一定の間隔になるようにすることで高速化が可能であると考える．これにより，あるベースアドレスにスタックの大きさとスレッドIDを掛けたものの和をとることで各々のスタックのベースアドレスを求めることが可能になり，探索等の複雑な処理の必要がなくなり，高速化が図れると考える．

（3）スタック間の間隔をどの程度にとるか．

スタック間の間隔とは，スタックの大きさの上限のアドレスと次のスタックのベースアドレスとの差のことである．その間隔が無い，または狭い場合には，スタックが大きくなっていくと，他のスレッドのスタック領域を破壊する状況が頻繁に発生する可能性が存在する．また，間隔が広い場合には，設定したスタックの大きさを超える時に，スタックの大きさを拡張することが容易にできるが，メモリを無駄に使用することも多くなる．

他のスタック領域に侵入しようとした場合の対処方法としては，以下に述べる2つの方法が考えられる．

1）侵入しようとしたスレッド，またはそれが属するプロセスを異常終了させる．

2）スタック領域を拡張して，侵入を防ぐ．

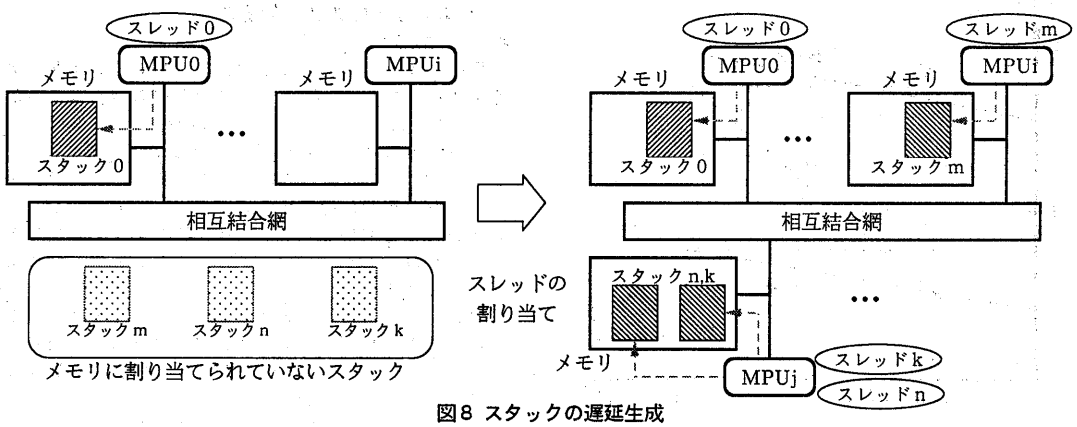


図8 スタックの遅延生成

1）の方法では，スタック内容の破壊に対する保証はできるが，異常終了が頻繁に発生する可能性がある．この時には，確保するスタック領域の大きさを大きくすればよい．しかし，これは実行時にしか判明せず，使用者から見ると使いにくい．これから，2）の方法が良いと考えられる．しかし，この処理にはオーバヘッドが伴う．本システムでは，当初は実現が容易な1）をとる．これは，スタックの間隔を十分大きくとることで，あまりスタックを拡張する必要は発生しないと考えるからである．しかし，これについては，スレッドを用いたアプリケーションを実行してその必要性を検討する必要がある．

4. 3. 2 スタック領域の遅延生成

スタックにアクセスするのはそれを所有しているスレッドのみであるから，ネットワークのトラフィックを減らすためにも，スレッドが実行しているプロセッサのローカルメモリ上にスタックを置く方がよい．しかし，スレッドが実行されるプロセッサは，スケジューリングにもよるが，スレッド生成時には判明していないことが多い．このため，スレッドの実行開始時までスタックの物理アドレスへのマッピングを遅らせ，スタックをローカルメモリ上にマッピングすれば高速なアクセスが実現できる（図8）．もちろん，仮想空間上にはスレッドのスタック領域は予め確保しておく必要がある．上記のスタック領域の遅延生成は，実行時に発生するページフォールトにより実行プロセッサを認識し，そのプロセッサ上で実際のスタック用の物理メモリのページ割り当て，スタックへの生成時に必要な情報の書き込み処理等を行うことにより実現できると考える．他の方法として，複数のスレッドのスタックをできるだけ異なるプロセッサ上に配置する方式が考えられる．これにより，複数を単一プロセッサ上に生成する場合よりメモリへのアクセス競合が軽減される．これはコモン空間のエントリの仮想アドレス空間への割り当て方法，つまり割り当てアルゴリズムにより行うことができる．本システムでは，上記のスタック領域の遅延生成の手法を用いる．この詳細な実現は現在検討中である．

5. おわりに

以上，開発中の並列OSのプロセス管理およびメモリ管理について述べた．現在Sun－4上にマルチスレッドが実現できる環境を構築しており，今後マルチプロセス環境を実現する予定である．

謝辞

我々と共に開発を進めている森，蒲池，廣谷，岩田，甲斐，上野，杉山の各氏，および日頃ご討論頂く富田研究室の皆様に感謝致します．

参考文献

〔1〕K.Murakami et al.："The Kyushu University Reconfigurable Parallel Processor －Design Philosophy and Architecture－",Proc. of IFIP 11th World Computer Congress,pp.995－1000 (1989).

〔2〕K.Murakami et al.："The Kyushu University Reconfigurable Parallel Processor －Design of Memory and Intercommunication Architectures －",Proc. of ACM SIGARCH Int'l Conf.on Supercomputing, pp.351－360 (1989).

〔3〕福田ほか：" 可変構造型並列計算機の並列／分散オペレーティング・システム", 情報処理学会オペレーティング・システム研究会, 89－OS－43－8 (1989).

〔4〕福澤ほか：" 可変構造型並列計算機のオペレーティング・システム －スレッドの実現－", 情報処理学会オペレーティング・システム研究会, 89－OS－45－6 (1989).

〔5〕蒲池ほか：" 統合型並列化コンパイラ・システム －ネットワークシンセシス－", 情報処理学会第40回全国大会講演論文集, 1G－2, pp.653－654 (1990)

〔6〕John K.Ousterhout et al.："Medusa：An Experiment in Distributed Operating System Structuer",Communications of the ACM , Vol. 25, No.2 , pp.397－409 (1980).

〔7〕草野ほか：" 可変構造型並列計算機のオペレーティング・システム －メモリ管理－", 情報処理学会第40回全国大会講演論文集, 6G－1, pp.740－741 (1990).