(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第4949231号
(P4949231)
(45)発行日　平成24年6月6日(2012.6.6)　(24)登録日　平成24年3月16日(2012.3.16)

(51)Int.Cl.　　F I
G06F 9/46 (2006.01)　　G06F 9/46 410
G06F 9/38 (2006.01)　　G06F 9/38 370X

請求項の数 12　(全 38 頁)

(21)出願番号　特願2007-506565(P2007-506565)
(86)(22)出願日　平成17年3月31日(2005.3.31)
(65)公表番号　特表2007-531167(P2007-531167A)
(43)公表日　平成19年11月1日(2007.11.1)
(86)国際出願番号　PCT/US2005/010923
(87)国際公開番号　WO2005/098624
(87)国際公開日　平成17年10月20日(2005.10.20)
審査請求日　平成18年9月28日(2006.9.28)
(31)優先権主張番号　10/816,103
(32)優先日　平成16年3月31日(2004.3.31)
(33)優先権主張国　米国(US)

前置審査

(73)特許権者　593096712
インテル　コーポレイション
アメリカ合衆国　９５０５２　カリフォルニア州　サンタ　クララ　ミッション　カレッジ　ブールバード　２２００
(74)代理人　100070150
弁理士　伊東　忠彦
(74)代理人　100091214
弁理士　大貫　進介
(74)代理人　100107766
弁理士　伊東　忠重
(72)発明者　グロショウスキ，エドワード，ティー
アメリカ合衆国　９５１１８　カリフォルニア州　サンノゼ　イェール　ドライヴ　５５６５

最終頁に続く

(54)【発明の名称】ユーザーレベルのマルチスレッドを提供する方法およびシステム

(57)【特許請求の範囲】
【請求項１】
第一のオペレーティング・システム（OS）生成スレッドを実行する段階であって、ここで、前記第一のOS生成スレッドは第一のシステム状態に関連付けられている、オペレーティング・システムによって生成されたスレッドである、段階と、
非特権ユーザーレベルプログラミング命令に遭遇する段階と、
非特権レベルプログラミング命令に反応して、第一のアプリケーション状態の第一の専有部分に関連付けられた第一の共有リソース・スレッド（シュレッド）を生成する段階であって、前記第一のシュレッドは、前記第一のアプリケーション状態の共有部分および前記第一のシステム状態を少なくとも第二のシュレッドと共有し、前記非特権レベルプログラミング命令は、OSの介入なしにハードウェアにおいてシュレッドを生成するためにハードウェアによって認識可能な命令セットアーキテクチャ（ISA）に含まれるものである、段階と、
前記ユーザーレベルプログラミング命令に反応して、前記第一のシュレッドを少なくとも前記第二のシュレッドと並行して実行する段階と、
前記第一のシュレッドと前記第二のシュレッドとの間で、前記第一のシュレッドおよび前記第二のシュレッドによって更新されることができるよう適応されている一つまたは複数の共有レジスタを介して通信する段階とを含み、前記一つまたは複数の共有レジスタは、前記第一のアプリケーション状態の共有部分および前記第一のシステム状態を共有するシュレッドの専有である、方法。

【請求項２】
前記第一のアプリケーション状態の前記第一の専有部分が、汎用レジスタ、浮動小数点レジスタ、MMXレジスタ、セグメント・レジスタ、フラグ・レジスタ、命令ポインタ、制御および状態レジスタ、SSEレジスタおよびMXCSRレジスタを含む複数のレジスタのうちの少なくとも一つと関連付けられていることを特徴とする、請求項１記載の方法。
【請求項３】
前記第一のシュレッドおよび前記第二のシュレッドが、現在の特権レベルを共有し、共通のアドレス変換を共有することを特徴とする、請求項１記載の方法。
【請求項４】
シュレッド破棄動作をエンコードする非特権ユーザーレベルプログラミング命令を受け取ることをさらに含むことを特徴とする、請求項１記載の方法。
【請求項５】
前記通信がユーザーレベルのシュレッド信号伝送命令を介して実行されることを特徴とする、請求項１記載の方法。
【請求項６】
ユーザーレベルのプログラミング命令に反応して、オペレーティング・システムの介入なしに前記第一のシュレッドおよび前記第二のシュレッドをスケジューリングすることさらに含むことを特徴とする、請求項１記載の方法。
【請求項７】
コンテキスト切り換え要求の受領に反応して、前記一つまたは複数のシュレッドに対応する一つまたは複数の状態を保存する、ことをさらに含むことを特徴とする、請求項１記載の方法。
【請求項８】
オペレーティング・システムの介入なしに、ユーザーレベルの例外ハンドラコードにより、前記第一のシュレッドの実行の間に発生した例外を扱う、ことをさらに含むことを特徴とする、請求項１記載の方法。
【請求項９】
ある命令セットアーキテクチャ（ISA）を実装するマイクロプロセッサであって、前記マイクロプロセッサは複数のOS生成スレッドを実行し、前記マイクロプロセッサはまた、前記複数のOS生成スレッドのうちのあるOS生成スレッドに関連付けられた複数の共有リソース・スレッド（シュレッド）を並行して実行し、前記複数のシュレッドのそれぞれは前記OS生成スレッド内のある専有状態および前記OS生成スレッドの共有状態に関連付けられており、前記シュレッドのそれぞれは、前記ISAのアプリケーション・プログラム命令に反応して前記マイクロプロセッサのハードウェアを利用して生成され、前記OS生成スレッドの共有状態の一部であるレジスタが、前記複数のシュレッドによって直接更新可能であるよう適応されており、それにより前記複数のシュレッドの間の通信を許容するとともに、前記OS生成スレッドの共有状態に関連付けられていないスレッドによってはアクセス可能でない、マイクロプロセッサと、
ユーザーレベルのマルチスレッド動作を許容するための、前記ISAの一つまたは複数の命令を記憶するメモリとを有することを特徴とする、
システム。
【請求項１０】
前記一つまたは複数の命令が、オペレーティング・システムの介入なしにシュレッドを生成する命令を含むことを特徴とする、請求項９記載のシステム。
【請求項１１】
前記一つまたは複数の命令が、オペレーティング・システムの介入なしにシュレッドを破棄する命令を含むことを特徴とする、請求項９記載のシステム。
【請求項１２】
前記ユーザーレベルのマルチスレッド動作が、同一のスレッドに関連付けられた二つ以上のシュレッドの並行的な実行を含むことを特徴とする、請求項９記載のシステム。

【発明の詳細な説明】
【技術分野】
【０００１】
本発明のここでの諸実施形態はコンピュータシステムの分野に関する。特に、ここでの諸実施形態はユーザーレベルのマルチスレッドを提供するための方法およびシステムに関する。
【背景技術】
【０００２】
マルチスレッドは、プログラムまたはOS（オペレーティングシステム）が命令のシーケンスを一時に二つ以上実行する機能である。各ユーザー（ここではユーザーは別のプログラムでもよい）からのプログラムまたはシステムサービスの要求は、別個の素性をもつスレッドとして追跡される。プログラムはそのスレッドに対する初期要求のために動作し、他の要求によって中断されるので、そのスレッドのための作業の状態は、その作業が完了するまで追跡される。
【０００３】
コンピュータ処理の種別には、単一命令ストリーム・単一データストリームが含まれるが、これは命令の単一のストリームを含む従来のシリアル式フォンノイマンコンピュータである。第二の処理種別は単一命令ストリーム・多重データストリーム（SIMD: single instruction stream, multiple data streams）処理である。この処理方式には、多重算術論理プロセッサおよび単一制御プロセッサが含まれる。算術論理プロセッサのそれぞれがデータに対して演算をロックステップで実行し、制御プロセッサによって同期される。第三の種別は多重命令ストリーム・単一データストリーム（MISD: multiple instruction streams, single data stream）処理であるが、これは同じデータストリームの流れを異なる命令ストリームを実行する諸プロセッサの線形配列を通じて処理することに関わる。第四の処理種別は、多重命令ストリーム・多重データストリーム（MIMD: multiple instruction streams, multiple data streams）処理であり、これは複数のプロセッサを使用し、そのそれぞれが該それぞれのプロセッサに供給されたデータストリームを処理するための独自の命令ストリームを実行する。MIMDプロセッサはいくつかの命令処理ユニット、複数の命令シーケンサ、よっていくつかのデータストリームを有しうる。
【発明の開示】
【発明が解決しようとする課題】
【０００４】
今日のマルチスレッド・マイクロプロセッサによって採用されているプログラミング・モデルは、伝統的な共有メモリ・マルチプロセッサと同じである：複数のスレッドがあたかも独立のCPUで走るかのようにプログラムされる。スレッド間の通信はメインメモリを通じて実行され、スレッド生成／破棄／スケジューリングはOSによって実行される。マルチスレッドはこれまで、プログラマーが直接スレッドにアクセスできるような、アーキテクチャ的に目に見える仕方では供給されていなかった。
【０００５】
本発明のここでの諸実施形態は、以下の詳細な記述を図面とともに参照することからより完全に理解され、認識されるであろう。
【課題を解決するための手段】
【０００６】
ユーザーレベルのマルチスレッドを提供する方法およびシステムが開示される。本技法に基づく方法は、命令セットアーキテクチャ（ISA: instruction set architecture）を介して一つまたは複数の共有リソース・スレッド（shared resource thread）（シュレッド［shred］）を実行するためのプログラミング命令を受け取ることを含む。一つまたは複数の命令ポインタがISAを介して構成設定され、前記一つまたは複数のシュレッドがマイクロプロセッサにより同時的に実行される。ここで、マイクロプロセッサは複数の

命令シーケンサを含んでいる。
【発明を実施するための最良の形態】
【０００７】
以下の記述では、説明の目的で個別的な述語が述べられる。しかし、こうした個別的な詳細が必須でないことは当業者には明らかであろう。以下の詳細な記述のいくつかの部分はアルゴリズムおよびコンピュータメモリ内のデータビットに対する演算の記号表現を用いて呈示されている。これらのアルゴリズム的記述および表現は、データ処理分野の当業者が自らの仕事の内容を他の当業者に最も効率的に伝達するために使っている手段である。アルゴリズムとはここでは、そして一般にも、所望の結果に導く一連の自己完結的な処理の系列であると考えられる。前記処理は物理量の物理的な操作を要求するものである。必須ではないが通例、そうした量は、保存、転送、組み合わせおよびその他の操作ができる電気信号または磁気信号の形をとる。時には主として慣用上の理由のため、これらの信号をビット、値、要素、記号、文字、項、数などと称することが便利であることがわかっている。
【０００８】
しかし、これらの、そして同様の用語はすべて適切な物理量と関連しており、単にそれらの量に適用される便利なラベルにすぎないことを留意すべきである。以下の議論から明らかなように、特別に断らない限り、本記載を通じて、「処理」または「計算」または「計数」または「決定」または「表示」などといった用語を用いた議論は、コンピュータシステムのレジスタおよびメモリ内の物理的な（電子的な）量として表現されているデータを操作し、コンピュータシステムメモリまたはレジスタまたはその他のそのような情報記憶、伝送もしくは表示装置内の物理的な量として同様に表現される他のデータに変換するコンピュータシステムまたは同様の電子計算装置の動作およびプロセスを言っていることが了解される。
【０００９】
本発明の諸実施形態はまた、ここの動作を実行するための装置にも関する。この装置は必要とされる目的のために特別に構築されたものでもよいし、あるいは汎用コンピュータであって該コンピュータ内に保存されるコンピュータプログラムによって選択的に活性化もしくは再構成設定されるのでもよい。そのようなコンピュータプログラムはコンピュータ可読記憶媒体に保存されうる。そのような媒体としては、これに限定されるものではないが、フロッピー（登録商標）ディスク、光ディスク、CD-ROM、光磁気ディスクを含む任意の種類のディスク、読み出し専用メモリ（ROM）、ランダムアクセスメモリ（RAM）、EPROM、EEPROM、磁気もしくは光カードまたは電子的な命令を保存するのに好適な任意の種類の媒体であってそれぞれコンピュータシステムバスに結合されているものがある。
【００１０】
ここに呈示される動作および表示はいかなる特定のコンピュータまたはその他の装置に本来的に関係しているものでもない。さまざまな汎用システムがここの思想に従ったプログラムとともに使用されうるし、あるいは必要とされる動作を実行するためにより特化した装置を構築することが便利であると判明することもありうる。多様なこれらのシステムのための必要とされる構造は以下の記述から明らかとなろう。さらに、本発明の実施形態はいかなる特定のプログラミング言語との関連で記載されているのでもない。ここに記載される本発明の諸実施形態の思想を実現するためには多様なプログラミング言語が使用されうることは理解されるであろう。
【００１１】
この明細書を通じて使用される「ユーザー」は、アプリケーション・プログラム、非特権コードおよび同様なソフトウェアといったユーザーレベル・ソフトウェアのことを述べている。ユーザーレベル・ソフトウェアはOSまたは同様の特権ソフトウェアからは区別される。本発明のある実施形態によれば、以下の記述は上述したMIMDプロセッサに適用される。
【００１２】

図１は、本発明のある実施形態に基づく、本方法および装置を利用する例示的なコンピュータシステム１００のブロック図を示している。コンピュータシステムはプロセッサ１０５を含む。チップセット１１０がシステム１００にメモリおよびI/O機能を提供する。より具体的には、チップセット１１０はグラフィックおよびメモリ・コントローラ・ハブ（GMCH: Graphics and Memory Controller Hub）１１５を含む。GMCH１１５は、プロセッサ１０５と通信するホストコントローラとしてはたらき、さらにメインメモリ１２０のためのコントローラとしてはたらく。本発明のある実施形態によれば、プロセッサ１０５はマルチスレッドのユーザーレベルへの拡張を可能にする。GMCH１１５はまた、それに結合されている先進グラフィックポート（AGP: Advanced Graphics Port）コントローラ１２５へのインターフェースをも提供する。チップセット１１０はさらに、数多くのI/O機能を実行するI/Oコントローラ・ハブ（ICH: I/O Controller Hub）を含む。ICH１３５はシステム管理バス（SMバス：System Management Bus）１４０に結合されている。

【００１３】

ICH１３５は、周辺コンポーネント相互接続（PCI: Peripheral Component Interconnect）バス１５５に結合されている。スーパーI/O（S/O: super I/O）コントローラ１７０がICH１３５に結合されて、キーボードおよびマウス１７５のような入力装置への接続性を提供する。汎用I/O（GPIO: general-purpose I/O）バス１９５がICH１３５に結合されている。USBポート２００が図のようにICH１３５に結合されている。プリンタ、スキャナ、ジョイスティックなどのようなUSBデバイスはこのバス上でシステム構成に追加されることができる。IDEドライブ２１０をコンピュータシステムに接続するために統合ドライブエレクトロニクス（IDE: integrated drive electronics）バス２０５がICH１３５に結合されている。論理的にはICH１３５は、単一の物理コンポーネント内の複数PCIデバイスのように見える。

【００１４】

プロセッサ１０５には命令セットアーキテクチャが含まれている。命令セットアーキテクチャ（ISA: instruction set architecture）はプロセッサ１０５のようなマイクロプロセッサの抽象モデルで、状態要素（レジスタ）およびそれらの状態要素を動作させる命令からなる。命令セットアーキテクチャは、マイクロプロセッサの振る舞いの抽象的な仕様を提供することによって、プログラマーおよびマイクロプロセッサ設計者の両方に対して、ソフトウェアとハードウェアの間の境界のはたらきをする。

【００１５】

シリコンチップ上で利用可能なトランジスタ数の向上は、汎用マイクロプロセッサへのマルチスレッドの導入を可能にした。マルチスレッドは２つの異なる仕方で実装されうる：チップレベル・マルチプロセッサ（CMP: chip-level multiprocessor）および同時マルチスレッド・プロセッサ（SMT: simultaneous multithreaded processor）である。このどちらもプロセッサ１０５として使用されうる。

【００１６】

図２は、本発明のある実施形態に基づく、例示的なチップレベル・マルチプロセッサを示している。プロセッサ２００のようなチップレベル・マルチプロセッサにおいては、複数のCPUコア２１０～２１３が単一のシリコンチップ２００に集積されている。CPUコア２１０～２１３のそれぞれは独立した実行スレッド２２０～２２３を実行できるが、一部のリソース（キャッシュなど）がCPUコア２１０～２１３のうちの二つ以上によって共有されていてもよい。

【００１７】

図３は、本発明のある実施形態に基づく、例示的な同時マルチスレッド・プロセッサ３００を示している。プロセッサ１０５はプロセッサ３００のような同時マルチスレッド・プロセッサであることができる。同時マルチスレッド・プロセッサ３００においては、単一のCPUコア３１０が、複数の実行スレッドを実行できる。CPUコア３１０は、極度に細かい粒度でCPUリソースを共有する（しばしば各リソースでどのスレッドを処理するかをク

ロックごとに決定する）ことによって、ソフトウェアには二つ以上のプロセッサのように見える。

【００１８】

図４は、本発明のある実施形態に基づく、例示的な非対称マルチプロセッサ４００を示している。プロセッサ１０５はマルチプロセッサ４００のような非対称マルチプロセッサであることができる。CPUコア４１０～４２７が異なるマイクロアーキテクチャを有するがISAは同じであるようなチップレベル・マルチプロセッサ４００を構築することが可能である。たとえば、少数の高性能CPUコア４１０～４１１が多数の低パワーCPUコア４２０～４２７と一緒に集積されてもよい。この種の設計は、高いスカラー性能とともに高い総スループットを実現できる。２つの種類のCPUコアはソフトウェアには、通常の共有メモリ・スレッドとして、あるいはシュレッドとして、あるいは両者の何らかの組み合わせとして見える。命令セットアーキテクチャ（ISA）はプロセッサ１０５のようなマイクロプロセッサの抽象モデルであり、状態要素（レジスタ）およびそれらの状態要素に作用する命令からなる。ISAはマイクロプロセッサの振る舞いの抽象的な仕様を提供することによって、プログラマーおよびマイクロプロセッサ設計者の両方に対して、ソフトウェアとハードウェアの間の境界のはたらきをする。本プログラミング・モデルは、アプリケーション・プログラムが、直接的に複数の非対称CPUコアを制御することを可能にする。

【００１９】

〈共有メモリ・プログラミング・モデル〉

従来のマルチスレッド・マイクロプロセッサは、従来の共有メモリ・マルチプロセッサ・システムと同じプログラミング・モデルを採用している。プログラミング・モデルは次のとおりである。マイクロプロセッサがOSに複数の実行スレッドを提供する。OSはこれらのスレッドを使って複数のアプリケーション（「プロセス」）を並行して走らせる、および／または単一のアプリケーションからの複数のスレッド（「マルチスレッド」）を並行して走らせる。いずれの場合にも、複数のスレッドはソフトウェアには独立したCPUのように見える。メインメモリはすべてのスレッドによって共有され、スレッド間の通信はメインメモリを通じて実行される。CPU内のハードウェアリソースも共有されうるが、共有はマイクロアーキテクチャによりソフトウェアからは隠されている。

【００２０】

伝統的な共有メモリ・マイクロプロセッサのプログラミング・モデルは広く理解され、多くのOSおよびアプリケーション・プログラムによってサポートされているものの、そのモデルはいくつかの不都合な点がある。それは：

１）スレッド間の通信がメインメモリを介して実行され、よって著しく遅い。キャッシングによって遅延の一部を軽減しうるが、共有を容易にするためにしばしばキャッシュラインをあるCPUコアから別のCPUコアに渡す必要がある。

２）スレッド間の同期がメモリベースのセマフォを使って実行され、よって著しく遅い。

３）スレッドの生成、破棄、サスペンドおよび再開がOSの介入を必要とし、よって著しく遅い。

４）CPUマルチスレッド機能の改良が上述したメモリ遅延およびOS遅延によって薄められてしまうため、マイクロプロセッサ発売元が最も効率的なマルチスレッド機能を提供できない。

【００２１】

〈マルチスレッド・アーキテクチャ拡張〉

従来システムに関する上に述べた理由のため、本方法およびシステムは、プロセッサのアーキテクチャを拡張して、マルチスレッド・アーキテクチャ拡張を通じてアーキテクチャ的に目に見えるマルチスレッド機能を含めるようにする。複数の同時的な実行スレッド、複数の命令ポインタ、単一の処理要素内のある種のアプリケーション状態（レジスタ）の複数のコピーが提供される。複数の実行スレッドは、既存の共有メモリ・スレッドとは

区別できるものであり、シュレッド（shred）すなわち共有リソース・スレッド（shared resource thread）と称される。

【００２２】

本マルチスレッド・アーキテクチャ拡張（multithreading architecture extensions）（その例は以下ではMAXと称される）は、既存のアーキテクチャ機能を含み、加えて複数の同時シュレッドをサポートすることになる。各シュレッドは独自の命令ポインタ、一般レジスタ、FPレジスタ、分枝レジスタ、述語レジスタおよびある種のアプリケーションレジスタを有している。シュレッドを生成および破棄するために非特権命令が生成される。シュレッド間の通信は、共有メモリに加えて共有レジスタを通じて実行される。本マルチスレッド・アーキテクチャ拡張は共有レジスタへの原子的なアクセスを保証するため、セマフォの必要性は低下する。さらに、本マルチスレッド・アーキテクチャ拡張は、インテル（登録商標）による３２ビットアーキテクチャのような３２ビットアーキテクチャと、やはりインテル（登録商標）による６４ビットアーキテクチャのような６４ビットアーキテクチャと、あるいは１６ビットアーキテクチャとさえも使用できる。

【００２３】

本発明のある実施形態に基づく、従来の共有メモリ・マルチプロセッサ・スレッドとシュレッドとの比較を次の表に示す。

【００２４】

表１

【００２５】

【表１】

動作	共有メモリ・ マルチプロセッサ・スレッド	マルチスレッド・ アーキテクチャ拡張
生成、破棄	OS コール	非特権命令
通信	共有のメモリ	共有のレジスタおよびメモリ
同期	メモリ・セマフォ	レジスタおよびメモリ・セマフォ。 共有レジスタが原子的な更新を 保証。
システム状態	各スレッドに特有な システム状態	全シュレッドで共有の システム状態

【００２６】

本マルチスレッド・アーキテクチャ拡張が従来のアーキテクチャ拡張とは根本的に異なっていることに気づかれるであろう。従来のアーキテクチャ拡張がより多くの命令およびより多くのレジスタ（状態）を提供していたのに対し、マルチスレッド・アーキテクチャ拡張はより多くの実行単位を提供する。

【００２７】

〈アプリケーションおよびシステム状態〉

プログラマーに見えるCPU状態は、２つの範疇に分けられる：アプリケーション状態およびシステム状態である。アプリケーション状態はアプリケーション・プログラムとOSの両方によって使用され、制御される一方、システム状態はOSによってのみ制御される。

【００２８】

図５は、本発明のある実施形態に基づく、ユーザーレベルのマルチスレッドを提供するための例示的な実行環境を示す。実行環境６００に含まれるレジスタのアプリケーション状態は次の表のようにまとめられる。

【００２９】

表２

【００３０】
【表２】

３２ビットアーキテクチャのアプリケーション状態	名称	幅
汎用レジスタ ６０５	EAX, EBX, ECX, EDX, EBP, ESI, DEI, ESP	32 ビット
浮動小数点レジスタ ６２５	ST0-7	80 ビット
セグメントレジスタ ６１０	CS, DS, ES, FS, GS, SS	16 ビット
フラグ・レジスタ ６１５	EFLAGS	32 ビット、あるビットはアプリケーション
命令ポインタ ６２０	EIP	32 ビット
FP 制御および状態レジスタ ６２６～６３１	CW ６２６, SW ６２７, TW ６２８, FP オペコード６２９, 命令ポインタ６３０, オペランドポインタ６３１	16 ビット, 16 ビット, 16 ビット, 11 ビット, 48 ビット, 48 ビット
MMX レジスタ ６３５	MM0-7	64 ビット、 エイリアス ST0-7
SSE レジスタ ６４０	XMM0-7	128 ビット
MXCSR レジスタ ６４５	MXCSR	32 ビット

【００３１】
ユーザーレベル・マルチスレッド・レジスタ６５０～６６５は、以下に、より詳細に述べる。
【００３２】
３２ビットアーキテクチャシステム状態は次のようにまとめられる。
【００３３】
表３
【００３４】

【表３】

３２ビットアーキテクチャのアプリケーション状態	名称	幅
制御レジスタ ６２６	CR0-CR4	32 ビット
フラグ・レジスタ ６１５	サブセット EFLAGS	32 ビット、サブセット
メモリ管理レジスタ	GDTR, IDTR	48 ビット
ローカル記述子表レジスタ、タスクレジスタ	LDTR, TR	16 ビット
デバッグ・レジスタ	DR0-DR7	32 ビット
モデル固有レジスタ ６５０	MSR0-MSRN	64 ビット タイムスタンプカウンタ、APIC、機械チェック、メモリタイプ範囲レジスタ（memory type range registers）、パフォーマンス監視のためのレジスタを含む
共有レジスタ ６５５	SH0-SH7	32 ビット
シュレッド制御レジスタ ６６０	SC0-SC4	32 ビット

各シュレッドについて、アプリケーション状態は２つの範疇に分けられる：シュレッド毎アプリケーション状態および共有アプリケーション状態である。ここで述べるMAXプログラミング・モデルはシュレッド毎アプリケーション状態の特有なインスタンスを提供する一方、共有アプリケーション状態は複数のシュレッドの間で共有される。システム状態のコピーは一つしかなく、所与のスレッドに対応するすべてのシュレッドが同じシステム状態を共有する。アプリケーションおよび状態の近似的な分割は次の表に呈示されている。

【００３５】

表４

【００３６】

【表４】

状態	タイプ
汎用レジスタ（プログラム可能なサブセット） 浮動小数点レジスタ（プログラム可能なサブセット） SSE レジスタ（プログラム可能なサブセット） 命令ポインタフラグ（アプリケーションサブセット）	シュレッド毎の占有状態
汎用レジスタ（プログラム可能なサブセット） 浮動小数点レジスタ（プログラム可能なサブセット） SSE レジスタ（プログラム可能なサブセット） 共有レジスタ（新） フラグ（システムサブセット） メモリ管理レジスタ アドレス変換（TLB） 現在特権レベル 制御レジスタ	複数のシュレッドの間で共有され、各スレッドにとって占有
メインメモリ	複数のスレッドの間で共有

【００３７】
本マルチスレッド・アーキテクチャ拡張は、ほとんどのアプリケーション状態のプログラム可能な共有（sharing）または専有（privacy）を提供し、ソフトウェアが最良の分割を選択できる。プログラムはビットベクトルによって行われ、個々のレジスタが共有または専有のいずれかとして選択されることができる。ハードウェアの名称変更手段が、ビットベクトルによって指定される共有プールまたは専有プールのいずれかからレジスタを割り当てることができる。
【００３８】
MAXの全体としての記憶要求は従来の同時マルチスレッド・プロセッサおよびチップレベル・マルチプロセッサより少ない。伝統的な共有メモリ・マルチプロセッサ・プログラミング・モデルを実装する同時マルチスレッド・プロセッサまたはチップレベル・マルチプロセッサではアプリケーションおよびシステム状態全体が複製される必要があるのに対し、MAXではシュレッド毎の専有アプリケーション状態だけが複製される。
【００３９】
〈シュレッド／スレッド階層〉
各共有メモリ・スレッドは複数のシュレッドからなる。シュレッドと共有メモリ・スレッドは２レベルの階層をなす。別の実施形態では、３レベルの階層が共有メモリMAXプロセッサのクラスタから構築できる。クラスタはメッセージ渡しを使って通信する。OSがスレッドのスケジューリングを扱い、一方、アプリケーション・プログラムがシュレッドのスケジューリングを扱う。任意の所与のシュレッドから見た他のシュレッドがローカルまたはリモートであるという意味において、シュレッドは非一様である。シュレッド毎アプリケーション状態は各シュレッドについて複製される。共有アプリケーションおよびシステム状態はローカルなシュレッドに共通であり、各共有メモリ・スレッドについて複製される。メモリ状態は一つのコピーのみを有する。
【００４０】
図６は、本発明のある実施形態に基づく、シュレッドと共有メモリ・スレッドとの例示的な関係を示している。シュレッド毎アプリケーション状態５１０は各シュレッドについて複製される。共有アプリケーションおよびシステム状態５２０はローカルなシュレッドに共通であり、各共有メモリ・スレッドについて複製される。メモリ状態５３０は一つのコピーのみを有する。
【００４１】
MAXプログラミング・モデルではシステム状態５２０が複数のシュレッドの間で共有されるため、該複数のシュレッドは同じプロセスに属する。本マルチスレッド・アーキテクチャ拡張は、マルチスレッド・アプリケーション、ライブラリおよび仮想マシンによって使用されることが意図されている。MAXプログラミング・モデルはこの種のソフトウェアに、そのシュレッドに対する先例のない度合いの制御および共有メモリでは実現できないパフォーマンス潜在力を与える。
【００４２】
シュレッドはみな同じ特権レベルで走り、同じアドレス変換を共有するので、シュレッド間では保護確認は必要とされない。よって、伝統的な保護機構はシュレッド間通信の間には回避しうる。
【００４３】
共有システム状態のため、同じスレッド上で異なるプロセスを走らせるためにMAXプログラミング・モデルを使うことはできない。このため、MAXプログラミング・モデルおよび従来の共有メモリ・プログラミング・モデルは同じシステム内で共存する。
【００４４】
所与のCPUは有限個の物理シュレッドを提供するため、ソフトウェアは利用可能なシュレッド数を、ハードウェアスレッドの可視化と同様の仕方で可視化する。可視化の結果は、有限個の、並行して走っている物理シュレッドとともに潜在的には限りない数の仮想シュレッドを与える。

【００４５】
〈システムコール〉
OSコールは、アプリケーション・プログラムからOSに制御を移し、コンテキスト切り換えを実行することにより、従来式の仕方で処理されてもよい。MAXアーキテクチャでは、一つの主要な相違は、どのシュレッドでOSをコールしても、所与のスレッドに関連するすべてのシュレッドの実行がサスペンドされるということである。同じスレッドに属するすべてのシュレッドの状態を保存し、復元するのはOSの責任である。
【００４６】
追加的な状態のため、コンテキスト切り換えのオーバーヘッドが増大する。コンテキスト切り換えのメモリ・フットプリントはシュレッド数に比例して増える。しかし、コンテキスト切り換え時間はあまり増加しない。各シュレッドが、他のシュレッドと並列に自分の状態を保存／復元するからである。コンテキスト切り換え機構は、複数のシーケンサを使った並列的な、状態の保存／復元を可能にする。OSそのものは複数のシュレッドを使用する。
【００４７】
OSをコールするコストが増大するため、OSによって実行されていたある種の機能性はアプリケーション・プログラムに移行される。この機能性には、スレッド維持およびある種の例外および割り込みの処理が含まれる。
【００４８】
システムコールを実行するための代替的な実施形態は、コンテキスト切り換えが高価になりつつある一方、スレッドが安価になりつつあるという観察に基づいている。この実施形態では、あるスレッドがOSを走らせるのに専用とされ、第二のスレッドがアプリケーション・プログラムを走らせるのに専用とされる。アプリケーション・プログラムのシュレッドがシステムコールを実行すると、それはOSシュレッドに（共有メモリを介して）メッセージを送り、応答メッセージを待つ。このようにして、メッセージ交換・待機機構が、従来式の制御移行・コンテキスト切り換え機構に取って代わる。どのスレッドのアドレス変換も変更は必要とされない。恩恵は、あるシュレッドによってOSに送られたメッセージがローカルな他のシュレッドを乱さないということである。

【００４９】
〈例外〉
従来のアーキテクチャでは、例外はアプリケーション・プログラムの実行をサスペンドし、OSの例外ハンドラを呼び出す。MAXプログラミング・モデルのもとでは、この振る舞いは望ましくない。あるシュレッドをサスペンドしてOSを呼び出すことは、（所与のスレッドに関連する）あらゆるシュレッドもサスペンドしてしまうからである。
【００５０】
この問題を解決するため、多くのタイプの例外を手当てする最初の機会をアプリケーション・プログラムに与える、新しいユーザーレベルの例外機構を導入する。ユーザーレベルの例外機構は、既存の例外タイプのいくつかが究極的にはアプリケーションそのものによって手当てされているとの観察に基づいている。
【００５１】
ユーザーレベルの例外機構のためには、例外がどう報告されるかが、例外がどう手当てされるかから区別される。例外は次のように３つの範疇に分けられる。
１．アプリケーション・プログラムに報告され、アプリケーション・プログラムによって手当てされる例外。たとえば、０による除算例外はその例外を引き起こしたアプリケーションに報告され、手当てもそのアプリケーションによってなされる。OSの関与は必要でもないし、望ましくもない。
２．アプリケーション・プログラムに報告され、その後、アプリケーション・プログラムが手当てのためにOSをコールする必要のある例外。アプリケーションによって起こされる

ページフォールトはアプリケーションに報告されてもよいが、ページ内でスワップするためにはアプリケーション・プログラムはOSをコールする必要がある。
３．OSに報告され、OSによって手当てされる必要のある例外。セキュリティ上の理由で、ハードウェア割り込みはOSに報告される必要がある。システムコール（ソフトウェア割り込み）は明らかにOSに報告される必要がある。

【００５２】

次の表は、上記の３つの各範疇の例外を示している。「キャッシュミスに際しての読み込み例外」および「微細粒度タイマー」の例外タイプは、本発明のある実施形態に関係した例外タイプとして与えられている。

【００５３】

表５

【００５４】

【表５】

例外タイプ	報告先	手当て主体
０による除算、オーバーフロー、バインド、FP 例外	アプリケーション	アプリケーション
アライメント確認	アプリケーション	アプリケーション
無効なオペコード	アプリケーション	アプリケーション
キャッシュミスに際しての読み込み例外	アプリケーション	アプリケーション
微細粒度のタイマー	アプリケーション	アプリケーション
スタックセグメントフォールト	アプリケーション	システム
一般保護	アプリケーション	システム
ページフォールト	アプリケーション	システム
二重フォールト	アプリケーション	システム
デバイス利用不能	アプリケーション	システム
ハードウェア割り込み	システム	システム
マスク不能割り込み	システム	システム
ソフトウェア割り込み(INTn)	システム	システム

【００５５】

アプリケーション・プログラムに報告された例外は、選択的にアプリケーション内で処理され、あるいは処理のためにOSに渡される。後者の場合、アプリケーション・プログラムは例外（ページフォールトのような）に応答してOSに手当てを明示的に要求するシステムコールを実行する。これは、暗黙のうちにOSがアプリケーションに代わってそのような手当てを実行する伝統的なアプローチとは対照的である。ネストされた例外を避けるため、例外をOSに中継するアプリケーション・コードはそれ自身が追加的な例外を招くことはないという特別な規定が設けられる。ユーザーレベルの例外機構は、影のレジスタセットにおけるCPUレジスタの最少数およびプロセッサ・ベクトルを固定位置に保存する。

【００５６】

〈仮想マシン〉

仮想マシンとマルチスレッド・アーキテクチャ拡張のここでの諸実施形態とは、互いに制約を課す。仮想マシンは、ソフトウェアが仮想化されているリソースにアクセスしようとするときには常に例外を発生させ、例外処理はシュレッドに対して著しいパフォーマンス上の影響をもつ。

【００５７】

仮想マシンでは、特権命令の実行または特権プロセッサ状態へのアクセスが例外を発生

させる。例外は仮想マシン・モニタに報告される（そしてそれによって手当てされる）必要がある。MAXでは、OS（および仮想マシン・モニタ）によって手当てされる例外は、所与のスレッドに関連するあらゆるシュレッドをサスペンドさせる。仮想マシン・モニタは複数のシュレッドの存在を理解する。仮想マシンのアーキテクチャは、非特権命令およびプロセッサ・リソースについて発生される例外の数を最小化する。

【００５８】
〈デッドロック〉
MAXアーキテクチャではシュレッドがローカルな他のシュレッドによってサスペンドされることができるので、デッドロック回避が複雑になる。アプリケーション・ソフトウェアは、一つのシュレッドがOSに手当てされる例外またはシステムコールを被っても、ローカルなすべてのシュレッドをサスペンドしてしまうデッドロックが生じないことを保証する。
【００５９】
ローカルな（シュレッド間の）通信および同期は、リモートの（スレッド間の）通信および同期とは区別される。ローカルな通信は、共有レジスタ６５５（図５に示してある）または共有メモリのいずれかを使って実行される。リモートの通信は共有メモリを使って実行される。ローカルなデータ同期は原子的なレジスタ更新、レジスタ・セマフォまたはメモリ・セマフォを使って実行される。リモートのデータ同期はメモリ・セマフォを使って実行される。
【００６０】
ローカルおよびリモートのシュレッド制御（生成、破棄）はいずれもMAX命令を使って実行される。シュレッド制御は、wait()またはyield()のためにOSをコールすることはしない。それは所与のスレッド上のあらゆるシュレッドをサスペンドするという意図しない効果を有しうるからである。スレッド維持のために使われるOSコールは、ユーザーレベルのシュレッド・ライブラリへのコールで置き換えられる。シュレッド・ライブラリは今度は、必要に応じてスレッドを生成および破棄するためにOSをコールする。

【００６１】
〈シュレッドとファイバ〉
シュレッドは、従来のOSで実装されるファイバとは異なる。相違は次の表にまとめられている。
【００６２】
表６
【００６３】

【表６】

特性	ファイバ	シュレッド
生成	一つのスレッドが複数のファイバを生成しうる	一つのスレッドが複数のシュレッドを生成しうる
同時並行	一つのスレッドはどの時点でも一つのファイバを走らせられる	一つのスレッドは複数のシュレッドを同時に走らせられる
スケジューリング	ファイバのスケジューリングは、協力的マルチタスク機構を使ってソフトウェアにより行われる	シュレッドのスケジューリングは、同時マルチスレッドまたはチップレベル・マルチプロセシングを使ってハードウェアにより行われる
状態	各ファイバは独自の占有アプリケーション状態をもつ	各シュレッドは独自の占有アプリケーション状態をもつ
状態保存	現在走っているファイバの状態がレジスタに保存される。アクティブでないファイバの状態はメモリに保存される。	現在走っている物理的シュレッドの状態のそれぞれがオンチップレジスタに保存される。アクティブでない仮想シュレッドの状態はメモリに保存される。
状態管理	OS が現在走っているファイバの状態をコンテキスト切り換えに際して保存／復元する	OS があらゆるシュレッドのアプリケーション状態をコンテキスト切り換えに際して保存／復元する

【００６４】

〈ハードウェア実装〉

マルチスレッド・アーキテクチャ拡張をサポートするマイクロプロセッサの実装は、チップレベル・マルチプロセッサ（CMP: chip-level multiprocessor）および同時マルチスレッド・プロセッサ（SMT: simultaneous multithreaded processor）の形をとることができる。従来のCMPおよびSMTプロセッサは、CPUリソースの共有をソフトウェアから隠そうとしている。これに対し、マルチスレッド・アーキテクチャ拡張のここでの諸実施形態を実装されたときには、プロセッサは共有をアーキテクチャの一部としてさらけ出す。

【００６５】

MAXプロセッサをチップレベル・マルチプロセッサとして実装するため、システム状態の複数のコピーをCPUコアどうしの間で同期状態に保つために、ブロードキャスト機構が使われる。共有されるアプリケーションおよびシステム状態のために高速通信バスが導入される。オンチップ通信はオフチップのメモリに比べて高速なため、これらの通信バスはMAXプロセッサに、共有メモリ・マルチプロセッサに対するパフォーマンス上の優位性を与える。

【００６６】

MAXプロセッサを同時マルチスレッド・プロセッサとして実装することは、ハードウェアがすでに必要なリソース共有を提供しているので可能である。MAXの実装を、マルチスレッドの３２ビットプロセッサ上でほとんど完全にマイクロコードで行うことも可能である。

【００６７】

ある実施形態によれば、本方法およびシステムは、複数のシュレッドの間でシステムコールおよび例外（OSに報告されるもの）の優先順位付けを可能にする。それにより、いかなる時点においても、一つのシュレッドの要求だけが手当てされる。システム状態は一時には一つのOSサービス要求しか扱えないので、優先順位付けおよび一つの要求の選択は必

要である。たとえば、シュレッド１およびシュレッド２が同時にシステムコールを行うとする。優先順位付け手段が、シュレッド１のシステムコールだけが実行されたがシュレッド２のシステムコールはまだ実行が始まっていないことを保証する。公正さへの配慮のため、優先順位付け手段はラウンドロビン選択アルゴリズムを採用するが、他の選択アルゴリズムを使ってもよい。

【００６８】
〈スケーラビリティ〉
MAXプログラミング・モデルのスケーラビリティは次によって決定される。
１）コンテキスト切り換えに際して保存／復元するのが実行可能である状態の量
２）コンテキスト切り換えの間に所与のスレッドに関連するあらゆるシュレッドをサスペンドすることから帰結する、並列度の減少
３）シュレッド間通信
シュレッド数が増加するにつれ、コンテキスト切り換えに際して保存／復元される必要のある状態の量が増加し、全シュレッドをサスペンドする結果として失われる潜在的な並列度が増加する。これら２つの因子が実際的なシュレッド数を制限することになる。
【００６９】
シュレッド間通信もスケーラビリティを制限する。該通信はオンチップリソースを使って実行されるからである。対照的に、伝統的な共有メモリ・マルチプロセッサ・モデルのスケーラビリティはオフチップ通信によって制限される。

【００７０】
〈共有分類〉
次の表には、シュレッドのアーキテクチャ、実装およびソフトウェア使用におけるさまざまな自由度での分類が呈示されている。
【００７１】
表７
【００７２】

【表７】

属性	オプション１	オプション２	オプション３
命令セット アーキテクチャ	均一……全シュレッドが同じ命令セットアーキテクチャを実装	不均一……諸シュレッドが異なる命令セットアーキテクチャを実装	
マイクロ アーキテクチャ 実装	対称的……全シュレッドが同じハードウェア・マイクロアーキテクチャで走る	非対称的……諸シュレッドが異なるハードウェア・マイクロアーキテクチャで走る	
アプリケーション 並列度	シーケンシャル……従来式の逐次コード	パラレル……並列化されたコード	
シュレッド生成	プログラマーが生成……シュレッドはプログラマーによって明示的に生成される	コンパイル……シュレッドはコンパイラによって自動的に生成される	固定機能……シュレッドはガーベッジ・コレクションのような特定の機能の専用
アーキテクチャの 正しさ	アーキテクチャ上……全シュレッドがプログラムのアーキテクチャの正しさに貢献	ヒント……いくつかのシュレッドはアーキテクチャ上の正しさに貢献するが、他のシュレッドはパフォーマンスにのみ貢献する	
入出力	計算。シュレッドは計算のみを行う。	I/O。シュレッドは計算に加えて入出力も行う。	

【００７３】

MAXアーキテクチャの２つの異なる種類が区別される：均一と不均一である。均一シュレッドは、すべてのシュレッドが同じ命令セットを実行するという点で均一マルチプロセッサと同様である。不均一マルチプロセッサと同様の仕方で、不均一シュレッドも可能である。たとえば、不均一シュレッドは：

・３２ビットプロセッサとネットワークプロセッサ、

・３２ビットプロセッサと６４ビットプロセッサ、

の間で構築されうる。

【００７４】

同様に、根底にあるマイクロアーキテクチャは対称的または非対称的のどちらでもありうる。後者の場合の例としては、若干の大規模な高性能CPUコアと多数の小型低パワーCPUコアを含む、図４に示したようなチップレベル・マルチプロセッサがある。

【００７５】

〈用途モデル〉

次の表は、本マルチスレッド・アーキテクチャ拡張の諸実施形態のためのいくつかの用途モデルをまとめている。

【００７６】

表８

【００７７】

【表８】

用途モデル	分類	説明	効果
プリフェッチ	均一ISA、シーケンシャルコード、コンパイラ生成、ヒント、計算	ヘルパー・スレッドがメイン・スレッドに先立ってメモリ位置をプリフェッチする。ヘルパー・スレッドはコンパイラによって生成される。	キャッシュミスでかなりの時間が使われるスカラーコードを高速化
従来式スレッドの置き換え	均一ISA、パラレルコード、プログラマー生成、アーキテクチャ上、計算	従来式の共有メモリ・スレッドの代わりにシュレッドが使われる。OSではなくライブラリがスレッドサービスを提供する。	スレッドのあるコードを高速化。スレッド・プリミティブは数桁速くなる。
コンパイラのための専用実行リソース	均一ISA、シーケンシャルコード、コンパイラ生成、アーキテクチャ上、計算	コンパイラがスカラーソースコードから複数のシュレッドを生成する。	コンパイラがシュレッドに対する直接的な制御を有する。
管理されたランタイム環境のための専用スレッド	均一ISA、固定機能、アーキテクチャ上、計算	シュレッドは管理されたランタイム機能専用とされる。たとえば、ジャストインタイム翻訳およびガーベッジ・コレクションは専用シュレッドを使って実行されうる。	翻訳およびガーベッジ・コレクションのシュレッドが本質的に無償になる。
並列プログラミング言語	均一ISA、パラレルコード、プログラマー生成、アーキテクチャ上、計算	複数のシュレッドにコンパイルされる並列コードをプログラマーが作成。	スレッド・プリミティブが命令として使われるのに十分な速さになる。
統合されたI/O機能をもつCPU	不均一ISA、パラレルコード、プログラマー生成、アーキテクチャ上、入出力	I/O機能がアプリケーション・プログラムによって直接実行される。たとえば、グラフィックおよびネットワーク処理。	I/O機能をCPUアーキテクチャに直接統合することを可能にする。
同時マルチISAのCPU	不均一ISA、非対称マイクロアーキテクチャ、プログラマー生成、アーキテクチャ上、計算	単一のCPUが複数のISA、たとえば３２ビットアーキテクチャと６４ビットアーキテクチャを実装。各ISAはプログラマーにはシュレッドとして利用可能。	興味深い可能性ではあるが、有用ではないかもしれない。
非対称コアマルチプロセッサ	均一ISA、非対称マイクロアーキテクチャ、アーキテクチャ上、計算	CMPが複数のコア、たとえば高性能および低パワーのコア混合を実装する。	良好なスカラーおよびスループット性能を実現。

【００７８】

〈プリフェッチ〉

プリフェッチ用途モデルでは、メイン・スレッドは一つまたは複数のヘルパー・スレッ

ドを生み、それがメインメモリからキャッシュラインをプリフェッチするのに使われる。ヘルパー・スレッドが生み出されるのは、メイン・スレッドでのキャッシュミスに応答してである。メインメモリへのアクセスは完了するのに数百から1000CPUクロックを必要とするので、キャッシュミスになった読み込みを失敗としてメインメモリに進むためのアーキテクチャ上の備えがされていない限り、スカラーコードの実行はメインメモリアクセスの間事実上停止することになる。

【００７９】
〈従来式スレッドの置き換え〉
シュレッドは、従来式スレッドの高性能な置き換えとして、マルチスレッド・アプリケーションによって使われることがありうる。ユーザーレベルのソフトウェア・ライブラリが、以前にOSによって実行されたシュレッド管理機能（生成、破棄など）を実行するために提供される。ライブラリは追加的なスレッドを要求するために、必要に応じてOSをコールするほか、シュレッド命令を使用する。ソフトウェア・ライブラリのコールは、コンテキスト切り換えが必要ないためOSコールよりもずっと高速である。

【００８０】
〈コンパイラのための専用実行リソース〉
コンパイラは、レジスタのようなほかのプロセッサ・リソースを使用するのと同じ仕方でシュレッドを使ってもよい。たとえば、コンパイラはプロセッサを８つの整数レジスタ、８つの浮動小数点レジスタ、８つのSSEレジスタおよび４つのシュレッドをもつものと見ることができる。シュレッドをリソースとして扱うことにより、コンパイラは、レジスタ割り当てと類似の仕方でシュレッドを割り当てる。レジスタと同様、アプリケーション・プログラムがハードウェアが提供するよりも多くの仮想シュレッドを要求する場合にシュレッドを補助記憶に散布／充填するための何らかの機構が必要である。従来のアーキテクチャでは、制御の流れは一つしかないので、通常プロセッサ・リソースとは見なされない。

【００８１】
〈管理されたランタイム環境のための専用スレッド〉
管理されたランタイム環境では、シュレッドはガーベッジ・コレクション、ジャストインタイム（just-in-time）コンパイルおよびプロファイリングのような機能の専用とされる。シュレッドはそのような機能を本質的に「無償で」実行する。シュレッドは命令セットアーキテクチャ（ISA）の一部として提供されているからである。ISAは、プロセッサのうち、プログラマーまたはコンパイラ作者に見える部分である。ISAはソフトウェアおよびハードウェアの間の境界のはたらきをする。

【００８２】
〈並列プログラミング言語〉
MAXは並列プログラミング言語およびハードウェア記述言語を直接サポートする。たとえば、iHDLまたはVerilogコンパイラは、ソースコードが明示的に並列であるため、直接的に複数シュレッドのためのコードを生成する。
【００８３】
チップレベル・マルチプロセッサによって可能となったスレッドの増殖はマルチスレッドのための言語サポートにつながる。そのようなサポートはOSおよびランタイム・ライブラリのコールを通じて提供される。マルチスレッドのための言語サポートはメインストリームの汎用プログラミング言語に移行される。

【００８４】
〈統合されたI/O機能をもつCPU〉

シュレッドはネットワーク・コプロセッサのようなI/O機能を実装するために使われる。シュレッドとして実装されるネットワーク・コプロセッサの一つの重要な相違点は、I/Oデバイスとしてではなく、CPUの一部として見えるということである。

【００８５】

従来システムでは、アプリケーション・プログラムが入出力を要求するとき、アプリケーション・プログラムはAPI（application program interface［アプリケーション・プログラム・インターフェース］）を使ってOSをコールする。OSは今度はデバイスドライバをコールして、そのデバイスドライバが要求をI/Oデバイスに送る。OSは複数のアプリケーション・プログラムからのI/O要求を待ち行列化またはシリアル化し、I/Oデバイスが一時には一つの（あるいは有限個の）要求のみを処理することを保証する役目を負う。これは、CPU状態が複数のアプリケーションの間で時間多重化されるのに対してI/Oデバイスの状態がシステムにとってグローバルであるため、必要なことである。

【００８６】

不均一シュレッドとして実装されるI/Oデバイスでは、I/Oデバイスの状態はCPUのアプリケーション状態の拡張として扱われる。アプリケーション・プログラムは、CPUのアプリケーション状態およびI/Oデバイス状態の両方を直接制御する。アプリケーション状態およびI/O状態の両方がコンテキスト切り換えに際してはOSによって保存／復元される。I/Oデバイスは、その状態がいくつかのアプリケーションの間で悪影響なしに時間多重化されることができるよう構成される。

【００８７】

〈同時マルチISAのCPU〉

６４ビットアーキテクチャが、３２ビットアーキテクチャ・アプリケーション・アーキテクチャならびに新しい６４ビット命令セットを「シームレス」として知られる機構を通じて含むよう定義される。３２ビットアーキテクチャの命令セットとの両立により、６４ビットアーキテクチャのプロセッサは新しい６４ビットアーキテクチャのアプリケーションのほか既存の３２ビットアーキテクチャのアプリケーションも走らせることができる。

【００８８】

現行の定義のもとでは、６４ビットアーキテクチャのCPUは、どの時点でも、６４ビットアーキテクチャのスレッドまたは３２ビットアーキテクチャのスレッドのいずれかを走らせる。２つのISAの間の切り換えは、６４ビットアーキテクチャbr.ia（３２ビットアーキテクチャへの分岐）および３２ビットアーキテクチャjmpe（６４ビットアーキテクチャへのジャンプ）を介して達成される。３２ビットアーキテクチャのレジスタは６４ビットアーキテクチャのレジスタにマッピングされるので、状態のコピーは一つしか必要とされない。

【００８９】

いかなる時点においても二つ以上の命令セットアーキテクチャが走っているマルチISAのCPUを創り出すことが可能である。これは、６４ビットアーキテクチャISAのためのシュレッドと３２ビットアーキテクチャISAのための第二のシュレッドを使うことによって達成しうる。均一シュレッドの場合のように、６４ビットアーキテクチャのシュレッドと３２ビットアーキテクチャのシュレッドの両方のために相異なるアプリケーション状態を提供する必要がある。６４ビットアーキテクチャのシュレッドと３２ビットアーキテクチャのシュレッドは同時に走る。

【００９０】

上記のマルチスレッド・アーキテクチャ拡張を通じたユーザーレベルのマルチスレッドを提供するための本方法およびシステムの諸特徴を述べたので、以下では３２ビットシステムのための実施形態を提供する。

【００９１】

〈３２ビットアーキテクチャの実施形態〉

IA-32アーキテクチャを参照しつつ記述しているが、ここに記載される方法およびシステムはIA-64アーキテクチャのような他のアーキテクチャにも適用されうることを読者は理解する。さらに、読者は、本発明のある実施形態に基づく例示的な実行環境を理解するために図５に戻るよう指示される。IA-32にユーザーレベルのマルチスレッド機能をもたらすために、いくつかのレジスタ６５０～６６０とともに、少数の命令が、IA-32のISAに追加される。

【００９２】

マルチスレッド・アーキテクチャ拡張は次の諸状態からなる：

・拡張を有効／無効にするためにOSまたはBIOSによって使われるモデル固有レジスタ６５０（MAX_SHRED_ENABLE）

・プロセッサが拡張を実装しているかどうかと利用可能な物理シュレッド数とを示す、CPUID拡張機能情報の３ビット

・各シュレッドが独自に専有アプリケーション状態のコピーを有するようにする、アプリケーション状態（EAX, EBXなど）のほとんどの複製

・シュレッド間の通信および同期のために使われうる、共有レジスタSH0-SH7　６５５の組

・シュレッド管理のために使われる制御レジスタSC0-SC4　６６０の組

【００９３】

このマルチスレッド・アーキテクチャ拡張は次の命令を有する。

・シュレッドの生成／破棄：forkshred、haltshred、killshred、joinshred、getshred

・通信：共有レジスタ６５５への／からの移動（mov）、共有レジスタ６５５への／からの同期移動

・同期（セマフォ）：cmpxshgsh、xaddsh、xchgsh

・信号伝送：signalshred

・マルチシュレッドモード（multi-shredded mode）への／からの遷移：entermsm、exitmsm

・状態管理：shsave、shrestore

・雑：シュレッド制御レジスタへの／からの移動

【００９４】

さらに、IA-32機構には以下の機能が設けられる。

・IA-32例外機構は、例外に際して（あてはまる場合には）マルチシュレッドモードを終了して全シュレッド状態を保存する

・IA-32のIRET命令は（あてはまる場合には）全シュレッド状態を復元してマルチシュレッドモードに戻る

・ユーザーレベルの例外機構が導入されている。

【００９５】

〈構成設定〉

モデル固有レジスタ（MSR: model specific register）６５０はマルチスレッド・アーキテクチャ拡張を可能にするために使われる。MSRは下記に記述される。

【００９６】

表９

【００９７】

【表９】

レジスタアドレス（十六進）	レジスタアドレス（十進）	レジスタ名フィールドおよびフラグ	共有／特有	ビットの説明
1F0H	496	MAX_SHRED_ENABLE	共有	ビット０がマルチスレッド・アーキテクチャ拡張を有効にする。リセット時に０に初期化される。OSまたはBIOSがこのレジスタに１を書き込むことによって明示的に MAX を有効にする必要がある。

【００９８】

シュレッドMSR６５０のようなモデル固有レジスタは、特権レベル０でのみ書き込みおよび読み出しが行われる。マルチスレッド・アーキテクチャ拡張が有効にされていなければ、旧来のコードの実行はシュレッド番号０に制限される。

【００９９】

表１０

【０１００】

【表１０】

初期の EAX の値	プロセッサについて与えられる情報
1H	EAX バージョン情報（型、ファミリー、モデル、ステッピング ID） EBX ビット 7-0: ブランド指数 ビット 15-8: CLFLUSH ラインサイズ（値 8＝キャッシュラインサイズはバイト単位） ビット 23-16: 物理プロセッサあたりの論理プロセッサ数 ビット 31-24: ローカルな APIC ID ECX 拡張機能情報 EDX 機能情報

【０１０１】

〈CPUID〉

IA-32のCPUID命令は、設けられている物理スレッド数の計数とともに、プロセッサがマルチスレッド・アーキテクチャ拡張をサポートしていることの指標を返すよう修正される。これはECXにおいて返される拡張機能情報に３ビット（NSHRED）を追加することによってなされる。CPUID命令によって返される情報は次の表に与えられている。

【０１０２】

表１１

【０１０３】

【表１１】

初期のEAXの値	プロセッサについて与えられる情報
1H	EAXバージョン情報（型、ファミリー、モデル、ステッピングID) EBX ビット7-0: ブランド指数 ビット15-8: CLFLUSHラインサイズ（値8＝キャッシュライン サイズはバイト単位） ビット23-16: 物理プロセッサあたりの論理プロセッサ数 ビット31-24: ローカルなAPIC ID ECX 拡張機能情報 EDX 機能情報

【０１０４】
表１２
【０１０５】
【表１２】

ビット番号	ニーモニック	説明
18:16	NSHRED	ハードウェアによってサポートされる物理シュレッド数を示す３ビット 000:１シュレッド／スレッド 001:２シュレッド／スレッド 010:４シュレッド／スレッド 011:８シュレッド／スレッド 100:１６シュレッド／スレッド 101:３２シュレッド／スレッド 110:リザーブ 111:リザーブ

【０１０６】
マルチスレッド・アーキテクチャ拡張が（MAX_SHRED_ENABLE MSRを通じて）有効にされていない場合、拡張機能情報はNSHREDについては000を返す。

【０１０７】
〈アーキテクチャ状態〉
マルチスレッド・アーキテクチャ拡張は、全状態を３つの範疇のうちの一つに置く。
・各シュレッドの専有
・ローカルなシュレッドの間で共有
・全シュレッドの間で共有
IA-32状態の前記範疇のそれぞれの内訳は上掲の表２に示されている。シュレッドの専有という状態は、シュレッドごとに一回繰り返される。シュレッドの専有という状態は、各シュレッドにとっての完全な専有である。特に、アーキテクチャは、よそのシュレッドからあるシュレッドの専有レジスタの読み出しや書き込みを個々に行ういかなる命令も提供しない。アーキテクチャが提供するshsaveおよびshrestore命令は、全シュレッドの専有状態のメモリへの読み出しおよび書き込みを集合的に行うためのものであるが、これらの命令は単一シュレッドモードにおいてしか実行されない。シュレッドの共有状態は上掲の表３に示されている。
【０１０８】
共有レジスタの組SH0-SH7 ６５５は、シュレッド間の通信および同期のために使われる。これらのレジスタ６５５は共有レジスタへのMOV命令および共有レジスタからのMOV命

令を通じて書き込みおよび読み出しが行われる。SH0-SH7レジスタ６５５は３２ビットの整数値を保存する。ある実施形態によれば、８０ビットの浮動小数点６２５および１２８ビットのSSEデータ６４０がメインメモリを通じて共有される。

【０１０９】

シュレッド制御レジスタの組SC0-SC4　６６０が提供される。これらのレジスタは次のように定義される。

【０１１０】

表１３

【０１１１】

【表１３】

レジスタ	名称	説明
SC0	シュレッド実行レジスタ	SC0はシュレッドごとに1ビットをもつビットベクトルを含む。ビット0がシュレッド0に、ビット1がシュレッド1に、などと対応する。各ビットは、対応するシュレッドが現在走っているか停止しているかを示す。マルチスレッド・アーキテクチャ拡張がMAX_SHRED_ENABLE　MSRを通じて無効にされているときは、SC0はシュレッド0のみが有効であることを示す値1を含む。
SC1	シュレッド実行割り込みレジスタ	マルチシュレッドから単一シュレッドモードに遷移する際にSC0の内容がSC1にコピーされ、単一シュレッドからマルチシュレッドモードに遷移する際にSC1の内容がSC0にコピーされる。
SC2	シュレッド状態保存／復元ポインタ	SC2はメモリ中でのシュレッド状態保存／復元領域をポイントする。このメモリ領域は、コンテキスト切り換えの際にすべての走っているシュレッドの状態を保存および復元するために使われる。
SC3	共有レジスタ空き／充満ビット	SC3は共有レジスタのための空き／充満のビットを含む。ビット0はsh0に、ビット1はsh1に、などと対応する。
SC4	ユーザーレベル割り込み表ベースアドレス	SC4はユーザーレベルの割り込み表のためのベースアドレスをポイントする。

【０１１２】

表１４

【０１１３】

【表１４】

ビット	型	複製	ニーモニック	説明
0	状態	Y	CF	桁上げフラグ（Carry flag）
2	状態	Y	PF	パリティフラグ（Parity flag）
4	状態	Y	AF	補助桁上げフラグ（Auxiliary carry flag）
6	状態	Y	ZF	零フラグ（Zero flag）
7	状態	Y	SF	符号フラグ（Sign flag）
8	システム	Y	TF	トラップフラグ（Trap flag）
9	システム	N	IE	割り込み有効フラグ（Interrupt enable flag）
10	制御	Y	DF	方向フラグ（Direction flag）
11	状態	Y	OF	オーバーフローフラグ（Overflow flag）
13:12	システム	N	IOPL	入出力特権フラグ（IO privilege flag）
14	システム	N	NT	ネストされたタスク（Nested task）
16	システム	N	RF	再開フラグ（Resume flag）
17	システム	N	VM	仮想86モード（Virtual 86 mode）
18	システム	N	AC	アライメント確認（Alignment check）
19	システム	N	VIF	仮想割り込みフラグ（Virtual interrupt flag）
20	システム	N	VIP	仮想割り込みペンディング（Virtual interrupt pending）
21	システム	N	ID	IDフラグ（ID flag）

【０１１４】
Yと記されたフラグはシュレッドごとに複製される。Nと記されたフラグは全シュレッドによって共有される単一のコピーを有する。
【０１１５】
３２ビットのEFLAGSレジスタ６１５は一群の状態フラグ、一つの制御フラグおよび一群のシステムフラグを含んでいる。プロセッサ１０５の初期化（RESETピンまたはINITピンをアサートすることによる）直後のEFLAGSレジスタ６１５は00000002Hである。このレジスタ６１５のビット１，３、５、１５および２２から３１まではリザーブされており、ソフトウェアはこれらのビットのいずれの状態も使ったり依存したりするべきではない。
【０１１６】
EFLAGSレジスタ６１５におけるフラグのいくつかは、専用の命令を使って直接修正できる。レジスタ全体を直接調査または修正できるようにするための命令はない。しかし、次の諸命令は、一群のフラグをプロシージャ・スタックまたはEAXレジスタへ、そしてそれらから移動させるのに使うことができる：LAHF、SAHF、PUSHF、PUSHFD、POPFおよびPOPFD。EFAGSレジスタ６１５の内容がプロシージャ・スタックまたはEAXレジスタに転送されたのち、フラグは、プロセッサのビット操作命令（BT、BTS、BTR、BTC）を使って調査および修正できる。
【０１１７】
タスクをサスペンドするとき（プロセッサのマルチタスク機能を使って）、プロセッサは自動的に、EFLAGSレジスタ６１５の状態を、サスペンドされるタスクのためのタスク状態セグメント（TSS: task state segment）に保存する。プロセッサは、自らを新しいタスクにバインドするとき、EFLAGSレジスタ６１５に新しいタスクのTSSからデータをロードする。
【０１１８】
割り込みまたは例外ハンドラ・プロシージャへのコールが行われるとき、プロセッサは自動的に、EFLAGSレジスタ６１５の状態をプロシージャ・スタックに保存する。割り込みまたは例外がタスク切り替えを用いて扱われるときは、EFLAGSレジスタ６１５の状態はサスペンドされるタスクのためのTSSに保存される。

【０１１９】
〈シュレッド生成／破棄〉
シュレッドはforkshred命令を使って生成されうる。書式は次のとおり。
【０１２０】
forkshred imm16, 目標IP
forkshred r16, 目標IP
２つの形が与えられている。一方はシュレッド番号を即値オペランド（immediate operand）としてもち、第二のものはシュレッド番号をレジスタオペランド（register operand）としてもつ。いずれの形についても、目標IPは即値オペランドとして指定され、その値は現在のIPではなく、コードセグメントの先頭（名目上０）を基準とする。
【０１２１】
forkshred imm16, 目標IPというエンコードは、長距離ジャンプ（far jump）命令と同様で、シュレッド番号が16ビットのセレクタの代わりに、目標IPが16/32ビットオフセットの代わりになっている。
【０１２２】
forkshred命令は適切な実行ビットをSC0に設定し、指定されたアドレスで実行を開始する。ユニックスのfork()システムコールとは異なり、forkshred命令は親シュレッドの状態をコピーしない。新しいシュレッドの実行は、他のすべての専有レジスタの現在の値とともに更新されたEIPを用いて開始される。新しいシュレッドがそのスタックをESPをロードすることによって初期化し、はいってくるパラメータを共有レジスタまたはメモリから取得するべきことが期待されている。forkshred命令は自動的にパラメータを渡すことはしない。
【０１２３】
目標シュレッドがすでに走っている場合、forkshredは#SNA（シュレッド利用不能）例外を発生させる。これはのちに述べるようにユーザーレベルの例外である。ソフトウェアは、すでに走っているシュレッドを開始させようとしないことを保証するか、あるいは代替的に、既存のシュレッドを停止させる#SNAハンドラを提供して改めてforkshredの実行に戻る。シュレッド番号がハードウェアによってサポートされる最大シュレッド数より大きい場合には、#GP(0)例外が発生する。
【０１２４】
現在のシュレッドの実行を終了させるためには、haltshred命令が使われる。haltshredはSC0における現在のシュレッドの実行ビットをクリアし、現在のシュレッドの実行を終わらせる。シュレッドの専有状態は停止中でさえも保持される。あるシュレッドが別のシュレッドの専有状態にアクセスするいかなる機構も存在しないので、停止したシュレッドの専有状態は見ることができない。しかし、その状態は持続し、そのシュレッドがforkshredを通じて再び実行を開始したときに見えるようになる。
【０１２５】
別のシュレッドの実行を途中で打ち切るためには、killshred命令が導入される。書式は次のとおり。
【０１２６】
killshred imm16
killshred r16
【０１２７】
ある実施形態によれば、シュレッド番号は１６ビットのレジスタまたは即値オペランドである。killshredはSC0における指定されたシュレッドの実行ビットをクリアして、そのシュレッドの実行を打ち切る。停止中、そのシュレッドの専有状態は保持される。
【０１２８】
目標シュレッドが走っていない場合には、killshredは黙って無視される。この振る舞いは、killshredと通常に終了するシュレッドとの間の競合を避けるために必要である。k

illshredを実行したのち、ソフトウェアは、目標シュレッドがもはや走っていないことを保証される。シュレッドはhaltshredを実行する代わりに自らを打ち切る（kill）こともできる。シュレッド番号がハードウェアによってサポートされる最大シュレッド数より大きい場合には、#GP(0)例外が発生する。

【０１２９】

指定されたシュレッドが終了する（SC0ビットがクリアされることで示される）まで待つため、joinshred命令が導入される。書式は次の通り。

【０１３０】

joinshred imm16
joinshred r16

【０１３１】

目標シュレッドが走っていなければ、joinshredはすぐに戻る。この振る舞いが、joinshredと通常に終了するシュレッドとの間の競合を避ける。joinshredを実行したのち、ソフトウェアは、目標シュレッドがもはや走っていないことを保証される。シュレッドがjoinshredを自らに行うことも許される（役には立たないが）。シュレッド番号がハードウェアによってサポートされる最大シュレッド数より大きい場合には、#GP(0)例外が発生する。joinshred命令は自動的に戻り値を渡しはしない。

【０１３２】

シュレッドが自らのシュレッド番号を決定できるようにするため、getshred命令が導入される。書式は次のとおり。

【０１３３】

getshred r32

【０１３４】

getshredは現在のシュレッドの番号を返す。getshredは、シュレッド番号によって指定されるメモリ配列にアクセスするために使われうる。getshred zeroは目的レジスタの全ビットに書き込むために１６ビットシュレッド番号を拡張する。

【０１３５】

全シュレッド生成／破棄命令のためには、シュレッド番号はレジスタオペランドまたは即値オペランドのどちらで指定されてもよい。即値形の実行のほうがレジスタ形の実行よりも高速であることが期待される。シュレッド番号が実行時ではなくデコード時において得られるであろうからである。即値形では、コンパイラがシュレッド番号を割り当てる。レジスタ形ではランタイムの割り当てが使用される。

【０１３６】

次の表はシュレッドの生成／破棄命令のまとめを呈示している。

【０１３７】

表１５

【０１３８】

【表１５】

命令	説明
forkshred imm16, 目標 IP forkshred r16, 目標 IP	指定されたアドレスでシュレッド実行を開始する
haltshred	現在のシュレッドを終了させる
killshred imm16 killshred r16	指定されたシュレッドを終了させる
joinshred imm16 joinshred r16	指定されたシュレッドが終了するまで待つ
getshred r32	現在のシュレッドの番号を返す

【０１３９】

forkshred、haltshred、killshred、joinshred、getshred命令はいかなる特権レベルで実行されてもよい。既存のIA-32のhlt命令が特権命令であるのに対して、haltshredは非特権命令である。

【０１４０】

killshredまたはhaltshredの実行の結果走っているシュレッドが０個になる可能性がある。この状態（SC0に0）は既存のIA-32の停止状態とは異なる。SC0は認められている状態である。ただし、ユーザーレベルのタイマー割り込みが生成されるまで有用ではない。

【０１４１】

〈通信〉

シュレッドは互いとの通信を、既存の共有メモリを通じて、およびその目的のために特別に導入されたレジスタの組を通じて行う。共有レジスタSH0-SH7　６５５は同じスレッドに属するすべてのローカルシュレッドによってアクセス可能である。SH0-SH7レジスタ６５５は、はいってくるパラメータをシュレッドに渡し、シュレッドからの戻り値を通信し、セマフォ動作を実行するために使われうる。各目的のために特定の共有レジスタ６５５はソフトウェアの慣例によって割り当てられる。

【０１４２】

各共有レジスタ６５５は対応する空き／充満ビットをSC3に有している。共有レジスタ６５５に書き込みおよび読み出しをするためには、共有レジスタ６５５にMOV、および共有レジスタ６５５からMOVの命令が使われる。これらは次のようにまとめられる。

【０１４３】

```
mov  r32,sh0-sh7
mov  sh0-sh7,r32
```

【０１４４】

命令のエンコードは既存の制御レジスタ６６０への／からのMOV、およびデバッグ・レジスタへの／からのMOVの命令と同様である。共有レジスタへの／からのMOV命令はいかなる特権レベルで実行されてもよい。これらの命令は、ソフトウェアが追加的な命令を使って同期を明示的に実行することを想定している。共有レジスタへの／からのmovは、SC3の空き／充満ビットの状態を調べることも、修正することもしない。

【０１４５】

共有レジスタ６５５へのMOV、および共有レジスタ６５５からのMOVの遅延は、共有メモリへのロードおよび保存の遅延より低いことが期待される。ハードウェア実装は、共有レジスタ６５５を見込みで読んで、他のシュレッド書き込みをせんさくすることがありそうである。ハードウェアは、共有レジスタ６５５に書き込むときには、強い順序付けの等価物を保証しなければならない。代替的な実施形態では、共有レジスタ６５５にアクセスするためにバリア命令が生成されることができる。

【０１４６】

あるアーキテクチャ特徴により、共有レジスタの順序付けとメモリの順序付けは互いから別個のものに保たれる。よって、あるシュレッドが共有レジスタ６５５に書き込み、次いでメモリ１２０に書き込む場合、共有レジスタ６５５の内容が共有メモリ内容より先に見える保証はない。この定義の理由は、不必要なメモリバリアを創り出すことなく、共有レジスタ６５５におけるループカウンタの高速アクセス／更新を可能にすることである。ソフトウェアが共有レジスタ６５５とメモリの両方にバリアを要求する場合、ソフトウェアは共有レジスタ・セマフォをメモリ・セマフォとともに両方実行する。メモリ・セマフォはバリアとしてはたらくことのほかは冗長である。

【０１４７】

同期のほかに迅速な通信を提供するため、共有レジスタへの／からの同期mov命令が使われる。これらの命令は次のようにまとめられる。

【０１４８】

```
syncmov  r32,sh0-sh7
```

syncmov sh0-sh7,r32

【０１４９】

命令エンコードは、既存の制御レジスタ６６０への／からのMOV、およびデバッグ・レジスタへの／からのMOVの命令とパラレルである。共有レジスタ６５５への同期movは、共有レジスタ６５５への書き込みの前に空き／充満ビットが空きを示すまで待つということのほかは、その非同期の対応物と同様である。共有レジスタ６５５への書き込み後、空き／充満ビットは充満に設定される。共有レジスタ６５５からの同期movは、共有レジスタ６５５から読む前に空き／充満ビットが充満を示すまで待つということのほかは、その非同期の対応物と同様である。共有レジスタ６５５から読んだのち、空き／充満ビットはクリアされて空きにされる。

【０１５０】

空き／充満ビットは、下記に述べるようにSC3への移動を用いて初期化されうる。共有レジスタへの／からの同期MOV命令はいかなる特権レベルで実行されてもよい。共有レジスタ通信命令は次のようにまとめられる。

【０１５１】

表１６

【０１５２】

【表１６】

命令	説明
mov r32, sh0-sh7	共有レジスタからの移動
mov sh0-sh7,r32	共有レジスタへの移動
syncmov r32, sh0-sh7	共有レジスタからの同期移動
syncmov sh0-sh7, r32	共有レジスタへの同期移動

【０１５３】

〈同期〉

一組の同期プリミティブが共有レジスタ６５５に作用する。同期プリミティブは、メモリでなく共有レジスタ６５５に作用するということのほかは、既存のセマフォ命令と同様である。命令は次の通り。

【０１５４】

表１７

【０１５５】

【表１７】

命令	説明
cmpxchgsh sh0-sh7, r32	共有レジスタをr32と比較。等しければZFがセットされてr32が共有レジスタにロードされる。等しくなければZFをクリアして共有レジスタをEAXにロードする。
xaddsh sh0-sh7, r32	共有レジスタをr32と交換。次いでr32を共有レジスタに加える。この命令は、原子的な動作を可能にするためにLOCKプレフィックスとともに使われうる。
xchgsh sh0-sh7, r32	共有レジスタをr32と交換。この命令は常に原子的である。

【０１５６】
同期プリミティブはいかなる特権レベルでも実行される。これらの命令は、SC3の空き／充満ビットの状態を調べることも、修正することもしない。

【０１５７】
〈マルチシュレッドモードの開始／終了〉
MAXアーキテクチャはマルチシュレッドモードと単一シュレッドモードの間の切り換えをする機構を提供する。単一シュレッドモードは、プロセッサが、一つを除いたすべてのシュレッドの実行を停止することによって秩序だった仕方でコンテキスト切り換えを実行できるようにする。SC0は現在の動作モードを次のように示す：
・どのビット位置であれちょうど一つの1を含むSC0は単一シュレッドモードを含意する。
・どのビット位置であれ一つの1という以外のパターンを含むSC0はマルチシュレッドモードを表す。
【０１５８】
コンテキスト切り換えを実行するためには次のことが必要である：
１）単一シュレッドモードに切り換えることによって一つを除いたすべてのシュレッドをサスペンドする。
２）シュレッド状態を保存する。
３）新しいシュレッド状態をロードする。
４）マルチシュレッドモードに切り換えることによってすべてのシュレッドの実行を再開する。
【０１５９】
マルチシュレッドモード（multi-shredded mode）および単一シュレッドモードに切り換えるためには、それぞれentermsmおよびexitmsmが使われる。entermsmはマルチシュレッドモードにはいるために使われる。この命令の実行に先立って、全シュレッドの状態がロードされる必要がある。entermsmはSC1の新しいシュレッド実行ベクトルをSC0にコピーする。entermsmは次いで指定された諸シュレッドを開始させる。
【０１６０】
entermsmの実行後、SC1の内容が全く追加的なシュレッドの実行をもたらさないことも可能である。この場合、プロセッサは単一シュレッドモードに留まる。entermsmを実行した結果として、entermsmが実行されたシュレッドがもはや走っていなくなることも可能である。マルチシュレッドモードを終了するにはexitmsmが使われる。exitmsmはSC0の現在のシュレッド実行ベクトルをSC1にコピーする。exitmsmを実行するシュレッドに対応するもの以外のすべてのSC0実行ビットはクリアされる。exitmsmを実行するシュレッド以外のすべてのシュレッドは停止される。これらの動作は原子的なシーケンスで実行される。SC0状態は単一シュレッドモードを示す。entermsmおよびexitmsmはいかなる特権レベルで実行されてもよい。

【０１６１】
〈状態管理〉
命令（shsaveおよびshrestore）は、集合的なシュレッド状態をそれぞれ保存および復元するため、すなわち全シュレッドの専有状態の内容をメモリに書き込むため、および全シュレッドの専有状態をメモリから読み出すために使われる。書式は次のとおり。
【０１６２】
shsave m16384
shrestore m16384
【０１６３】
メモリ保存領域のアドレスは、命令中の偏位によって指定される。アドレスは１６バイトの境界に整列させられる。メモリ保存領域は将来の拡張を許容するため、16キロバイトである。メモリ保存領域は、既存のFXSAVE／FXRESTOR書式を整数レジスタを追加すること

によって拡張する。各シュレッドのためのメモリ保存領域は次のように定義される。

【０１６４】

表１８

【０１６５】

【表１８】

オフセット	レジスタ
0-1	FCW
2-3	FSW
4-5	FTW
6-7	FOP
8-11	FIP
12-13	CS
14-15	リザーブ
16-19	FPU DP
20-21	DS
22-23	リザーブ
24-27	MXCSR
28-31	MXCSR_MASK
32-159	ST0-ST7
160-287	XMM0-XMM7
288-351	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
352-359	ES, FS, GS, SS
360-367	EIP
368-371	EFLAGS

【０１６６】

全シュレッドの内容は次式で与えられるアドレスに保存／復元される：

アドレス＝512×（シュレッド番号）＋（ベースアドレス）

メモリ保存領域は現在走っているシュレッドのEIPおよびESPを含む。shsaveは現在のEIPおよびESPをメモリに書き込む。分岐を避けるため、shrestore命令は現在のシュレッドのEIPやESPを上書きすることはしない。shrestore関数は、IRETの一部として実行されたとき、現在のシュレッドのEIPおよびESPを上書きする。

【０１６７】

shsaveおよびshrestoreはいかなる特権レベルで実行されてもよいが、単一シュレッドモードにあるときのみである。マルチシュレッドモードにあるときにshsaveまたはshrestoreが試みられると、#GP(0)例外が発生する。実装では、shsave／shrestoreの保存／ロード動作を実行するために利用可能な全ハードウェアリソースを自由に使用できる。

【０１６８】

shrestoreは無条件に、全シュレッドの状態をメモリからロードする。この振る舞いは、シュレッドの専有状態が一つのタスクから次のタスクへ漏れないことを保証するために必要である。shsaveは、無条件にまたは条件付きで、全シュレッドの状態をメモリに保存する。ある実装では、専有状態が修正されなかった場合にshsaveの保存動作の一部または全部をスキップするよう非アーキテクチャ的可視のダーティー・ビット（dirty bits）を維持しうる。

【０１６９】

shsaveおよびshrestore命令はシュレッドの専有状態のみを保存および復元する。共有レジスタ６５５を保存および復元することはOSの責任である。

【０１７０】

〈シュレッド制御レジスタ６６０への／からの移動〉

シュレッド制御レジスタSC0-SC4　６６０に書き込み、そこから読み出すための命令が提供されている。それは次のようにまとめられる。

【０１７１】

mov　r32,sc-sc4

mov　sc0-sc4,r32

命令エンコードは既存の制御レジスタ６６０への／からのMOV、およびデバッグ・レジスタへの／からのMOVの命令と同様である。シュレッド制御レジスタへの／からのMOV命令はいかなる特権レベルで実行されてもよい。悪意のあるアプリケーション・プログラムがシュレッド制御レジスタに書き込むことによって自分以外の何らかのプロセスに影響できないことを保証するための安全措置が設けられる。

【０１７２】

アプリケーション・プログラムは、SC0の内容を直接操作するのではなく、forkshredおよびjoinshredを使う。exitmsmはマルチシュレッドモードから単一シュレッドモードへの遷移を原子的な仕方で行う。現在のシュレッド実行状態を読むためにSC0からのmovを使い、次いでシュレッド実行状態を書き込むためにSC0へのmovを使うことのでは、シュレッド実行状態が読み出しと書き込みの間で変化しうるので、所望の結果が得られないのである。

【０１７３】

〈OS例外〉

MAXは、IA-32例外機構のためにいくつかの関わりを有する。まず、ユーザーレベルの例外機構により、いくつかの型の例外がそれを発生させたシュレッドに直接報告されることが可能になる。この機構については後述する。

【０１７４】

次に、IA-32例外機構は、コンテキスト切り換えを必要とする例外が存在する場合に複数のシュレッドを適正に処理するよう修正される。従来のIA-32例外機構に関する一つの問題は、ちょうど一つの実行中スレッドのためにCS、EIP、SS、ESP、EFLAGSを自動的に保存および復元するよう定義されているということである。

【０１７５】

既存のIA-32例外機構は、entermsm、exitmsm、shsave、shrestore命令の機能性を含むよう拡張される。コンテキスト切り換えを必要とする割り込みまたは例外が発生させられると、例外機構は次のことをする：

１）exitmsmを実行することによりマルチシュレッドモードを終了する。exitmsmは、その割り込みまたは例外を引き起こしているもの以外のすべてのシュレッドを停止させる。OSはその割り込みまたは例外を引き起こしたシュレッドを使ってはいられる。

２）SC2によって与えられる開始アドレスでshsaveを実行することにより、全シュレッドの現在の状態をメモリに保存する。

３）現在定義されているようなIA-32コンテキスト切り換えを実行する。

【０１７６】

マルチシュレッドプログラムに戻るためには、修正されたIRET命令は次のことを実行する：

１）現在定義されているようなIA-32コンテキスト切り換えを実行する。

２）SC2によって与えられる開始アドレスでshrestoreを実行することにより、全シュレッドの現在の状態をメモリから復元する。これはIA-32コンテキスト切り換えにおいて保存されたEIPおよびESPを上書きする。

３）entermsmを実行することによりマルチシュレッドモードにはいる。SC1の状態によっては、entermsmの実行は、プロセッサをして単一シュレッドモードに留まらせることもある。

【０１７７】

OSは、IRETを実行するのに先立って、メモリ中にシュレッド状態の保存／復元領域を設

定してそのアドレスをSC2にロードするよう要求される。OSはまた、SC1、SC3、SC4の状態を保存／復元することも要求される。

【０１７８】

複数のシュレッドがOSのサービスを必要とする例外に同時に遭遇する可能性がある。MAXアーキテクチャは一時には一つのOS例外しか報告できないので、ハードウェアは複数のシュレッドにわたるOS例外を優先順位付けし、ちょうど一つを報告して、他のすべてのシュレッドの状態を例外を発生させた命令がまだ実行されていない時点に設定する必要がある。

【０１７９】

〈ユーザーレベルの例外〉

MAXは、ある型の例外が完全にアプリケーション・プログラム内で処理されることを可能にする、ユーザーレベルの例外機構を導入する。いかなるOS関与、特権レベル遷移あるいはコンテキスト切り換えも必要でない。

【０１８０】

ユーザーレベルの例外が生起すると、次の未実行命令のEIPがスタックにプッシュされ、プロセッサは指定されたハンドラに差し向けられる。ユーザーレベルの例外ハンドラはそのタスクを実行し、次いで既存のRET命令を介して戻る。ある実施形態によれば、ユーザーレベルの例外をマスクするためにいかなる機構も設けられない。アプリケーションがユーザーレベルの例外を発生させるのは、該アプリケーションがそれを手当てする用意があるときのみであると想定されるからである。

【０１８１】

最初の二つのユーザーレベル例外を生成するために、２つの命令が提供される：signalshredおよびforkshredである。これらについて以下の節で述べる。

【０１８２】

〈信号伝送〉

signalshred命令は、指定されたシュレッドに信号を送るために使用される。書式は次のとおり。

【０１８３】

signalshred　imm16,　目標IP

signalshred　r16,　目標IP

目標シュレッドはレジスタオペランドまたは即値オペランドとして指定されうる。signalshred　imm16,　目標IPという命令のエンコードは、既存の長距離ジャンプ命令と同様で、シュレッド番号が16ビットのセレクタの代わりに、目標IPが16/32ビットオフセットの代わりになっている。長距離ジャンプの場合と同様、signalshredの目標IPは、現在のIPを基準にするのではなく、コードセグメントの先頭（名目上０）を基準として指定される。

【０１８４】

signalshredに応答して、目標シュレッドは、次の未実行命令のEIPをスタックにプッシュし、指定されたアドレスに差し向けられる。シュレッドは自分に信号を送ってもよく、その場合、効果は近距離コール（near　call）命令を実行するのと同じである。目標シュレッドが走っていない場合には、signalshredは黙って無視される。シュレッド番号がハードウェアによってサポートされる最大シュレッド数より大きい場合には、#GP(0)例外が発生する。

【０１８５】

signalshred命令はいかなる特権レベルで実行されてもよい。signalshred命令は目標シュレッドにパラメータを自動的に渡すことはしない。signalshredをブロックするためのいかなる機構も設けられていない。よって、ソフトウェアは、signalshredを発する前にブロック機構を実装するか、あるいはネスト可能なsignalshredハンドラを設けるかする

必要がありうる。

【０１８６】
〈シュレッド利用不能（SNA: Shred Not Available）〉
forkshredは、プログラムがすでに走っているシュレッドを開始させようとした場合には#SNA例外を発生させる。ソフトウェアの#SNAハンドラは既存のシュレッドに対してkill shredを実行して、forkshred命令に戻りうる。
【０１８７】
#SNA例外は、forkshred命令のEIPをスタックにプッシュし、SC4+0によって与えられるアドレスに差し向けることによって処理される。SC4+0のところにあるコードは実際のハンドラに分岐すべきである。例外ベクトルはSC4+16、SC4+32などに置かれる。ソフトウェアは、256とおりの可能なユーザーレベル例外をカバーするためにSC4+4095までのメモリをリザーブしている。メモリ／SC4機構内の割り込み表は、のちの時刻においてよりクリーンな機構で置き換えられる。

【０１８８】
〈サスペンド／再開とシュレッド可視化〉
マルチスレッド・アーキテクチャ拡張は、ユーザーレベルのソフトウェアが以下のような命令を使ってシュレッドをサスペンドまたは再開することを許容する。シュレッドをサスペンドするための命令：
１）メモリ中のシュレッド状態保存領域を初期化する。これは、サスペンド動作のためにアプリケーション・プログラムによって設定されたメモリ領域であって、SC2とポイントされるコンテキスト切り換えシュレッド状態領域とは異なる。
２）サスペンド・ハンドラをポイントするシュレッドに信号を送る。これはsignalshred 目標シュレッド， サスペンド・ハンドラIP を通じて行われる。
３）サスペンド・ハンドラが既存のmov， pusha， fxsave命令を使ってそのシュレッドの専有状態をメモリに保存する。
４）サスペンド・ハンドラがhaltshredを実行する。
５）オリジナルコードがjoinshredを実行してシュレッドが停止するまで待つ。
【０１８９】
サスペンド動作の時点でシュレッドがすでに停止されているかもしれないことが可能である。この場合、signalshredは無視され、サスペンド・ハンドラは呼び出されることがなく、joinshredが待つこともない。メモリ中のシュレッド状態保存領域はその初期値を保持するが、該初期値はすぐにhaltshredを実行するダミーシュレッドをポイントする必要がある。シュレッドを再開するには、逆の動作が実行される。
１）再開ハンドラをポイントするシュレッドをフォークする。これは、forkshred 目標シュレッド， 再開ハンドラIP を通じて行われる。
２）再開ハンドラが既存のmov， popa， fxrestor命令を使ってシュレッドの専有状態をメモリから復元する。
３）再開ハンドラが既存のRET命令を通じてシュレッドに復帰する。
【０１９０】
再開先のスレッドがすでに停止されているときには、再開ハンドラは、すぐにhaltshredを実行するダミーシュレッドにRETする。サスペンド／再開機能は、シュレッド可視化の可能性を開く。forkshredを実行する前に、ソフトウェアは同じシュレッド番号をもつ既存のシュレッドをサスペンドすることを選ぶことができる。joinshredを実行したのちに、ソフトウェアは同じシュレッド番号をもつ既存のシュレッドを再開することを選ぶことができる。サスペンド／再開シーケンスは再入可能ではないので、いかなる所与の時刻にもいかなる所与のシュレッドについても実行されているサスペンド／再開が一つだけであることを保証するための、ソフトウェアの決定的なセクションが必要である。これらの機構を使って、アプリケーション・プログラムが独自のプリエンプティブなシュレッド・ス

ケジューラを生成することが可能である。

【０１９１】

MAXの代替的な実施形態では、最初の利用可能なシュレッドを使ってフォークするための命令が存在する（allocforkshred r32）。ここでr32は割り当てられたシュレッド番号を用いて書かれる（forkshredではr32はフォークすべきシュレッド番号を指定する）。allocforkshredは、利用可能なハードウェア・シュレッドがあるかどうかを示すフラグをも返す。

【０１９２】

別の実施形態では、waitshred命令が、共有レジスタを使った待機同期を提供する（waitshred sh0-sh7, imm）。wait命令は待機機能を命令として提供する。この命令がなければ、次のようなループを使う必要がある。

【０１９３】

```
loop: mov eax, sh0
      and eax, mask
      jz loop
```

別の実施形態では、joinshredは、複数のシュレッド上で待機するためのビットマスクを与えられる。ビットマスクがなければ、joinshredは一つのシュレッドが終了するのを待ち、複数のシュレッド上で待機するためには複数のjoinshredが必要とされる。

【０１９４】

ある代替的な実施形態では、killshredは使用されない。signalshredとそれに続くjoinshredがkillshredの代わりに使用されうる。signalshredハンドラはhaltshred命令からなる。

【０１９５】

さらに別の実施形態では、forkshredとsignalshredを組み合わせることが可能である。forkshredとsignalshredは、シュレッドが現在走っているか停止しているかに関する振る舞いでしか違わない。signalshredが停止されているシュレッドを開始させることを許容されれば、signalshredは可能性としてはforkshredの代わりになることができる。

【０１９６】

図７は、本発明のある実施形態に基づく、ユーザーレベルのマルチスレッドの例示的なプロセスの流れ図である。次に述べるプロセスはアプリケーションまたはソフトウェア・プログラムによって開始されたと想定されている。次に述べるプロセスはいかなる特定のプログラムとの関連でもなく、上記の命令およびアーキテクチャによって達成されるユーザーレベルのマルチスレッドの一つの実施形態として記述される。さらに、次に述べるプロセスは、１６ビット、３２ビット、６４ビット、１２８ビットあるいはそれ以上のいずれのアーキテクチャであろうと、マルチプロセッサのようなマイクロプロセッサのISAとの関連で実行される。マルチプロセッサ（プロセッサ１０５のような）は、共有レジスタ、たとえば上掲の表３のレジスタの値を初期化する（処理ブロック７０５）。プロセッサ１０５はシュレッドを生成するforkshred命令を実行する（処理ブロック７１０）。複数の並行動作がプロセッサ１０５によって実行される。主（親）シュレッドがプロセッサ１０５によって実行される（処理ブロック７１５）。joinshred動作が実行されて、新しい目標シュレッドが実行を完了するのを待つ（処理ブロック７３０）。その間、新しい目標シュレッドはそのスタックを初期化し、はいってくるパラメータを共有レジスタおよび／またはメモリから取得し（処理ブロック７２０）、実行する（処理ブロック７２１）。現在の目標シュレッドの実行はhaltshred命令を使って終了される（処理ブロック７２３）。プロセッサ１０５は、実行結果を、シュレッドの実行結果が保存されているレジスタからプログラムまたはアプリケーションに返す（処理ブロック７３５）。ひとたびすべての実行されたデータが返されたら、プロセスは完了する（終了ブロック７９９）。

【０１９７】

ユーザーレベルのマルチスレッドを提供するための方法およびシステムが開示されている。本発明のここでの諸実施形態は、特定の例およびサブシステムに関連して述べられて

きたが、当業者には、本発明のここでの諸実施形態がそれらの特定の例またはサブシステムに限定されるものではなく、他の実施形態にも広がることは明らかであろう。本発明のここでの諸実施形態は、付属の請求項において規定される、これらの他の諸実施形態のすべてを含むものである。

【図面の簡単な説明】

【０１９８】

【図１】本発明のある実施形態に基づく、本方法および装置を利用する例示的なコンピュータシステムのブロック図である。

【図２】本発明のある実施形態に基づく、例示的なチップレベルのマイクロプロセッサを示す図である。

【図３】本発明のある実施形態に基づく、例示的な同時マルチスレッド・プロセッサを示す図である。

【図４】本発明のある実施形態に基づく、例示的な非対称マルチプロセッサを示す図である。

【図５】本発明のある実施形態に基づく、ユーザーレベルのマルチスレッド機能を提供するための例示的な実行環境を示す図である。

【図６】本発明のある実施形態に基づく、シュレッドと共有メモリ・スレッドとの間の例示的な関係を示す図である。

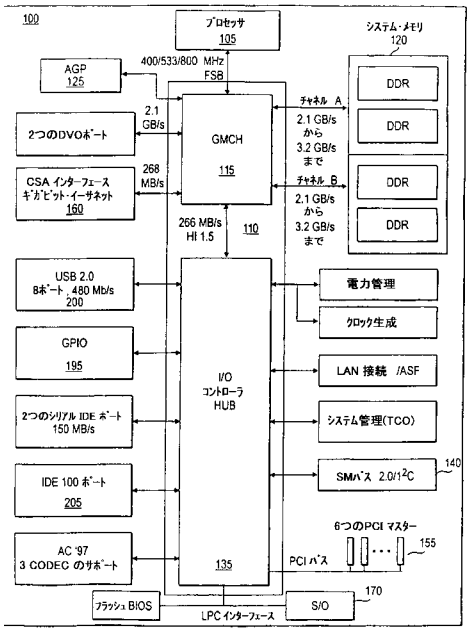
【図７】本発明のある実施形態に基づく、ユーザーレベルのマルチスレッド機能の例示的なプロセスの流れ図である。

【符号の説明】

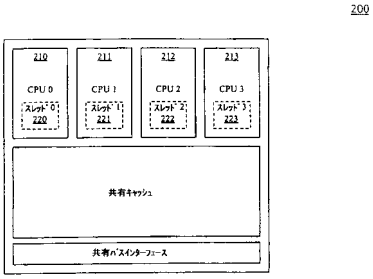
【０１９９】

７０１　開始
７０５　共有レジスタの値を初期化
７１０　forkshredを実行
７１５　主シュレッドを実行
７２０　共有レジスタから値を読む
７２１　シュレッドを実行
７２２　共有レジスタにシュレッド実行結果を書き込む
７２３　haltshredを実行
７３０　join動作を実行
７３５　シュレッド実行から返される値をレジスタから読む
７９９　終了

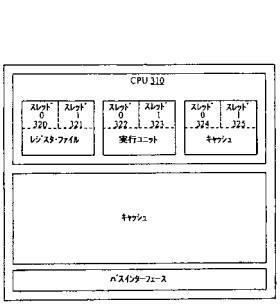
【図１】


100
プロセッサ
105
システム・メモリ
120
AGP
125
400/533/800 MHz
FSB
2.1
GB/s
2つのDVOポート
GMCH
115
チャネル A
2.1 GB/s
から
3.2 GB/s
まで
チャネル B
2.1 GB/s
から
3.2 GB/s
まで
DDR
DDR
DDR
DDR
CSA インターフェース
ギガビット・イーサネット
160
268
MB/s
266 MB/s
HI 1.5
110
USB 2.0
8ポート, 480 Mb/s
200
GPIO
195
2つのシリアル IDE ポート
150 MB/s
IDE 100 ポート
205
AC '97
3 CODEC のサポート
I/O
コントローラ
HUB
135
電力管理
クロック生成
LAN 接続 /ASF
システム管理(TCO)
SMバス 2.0/I2C
140
6つのPCI マスター
PCI バス
155
フラッシュ BIOS
LPC インターフェース
S/IO
170

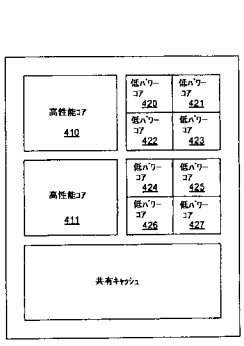
【図２】


200
210
CPU 0
スレッド 0
220
211
CPU 1
スレッド 1
221
212
CPU 2
スレッド 2
222
213
CPU 3
スレッド 3
223
共有キャッシュ
共有バスインターフェース

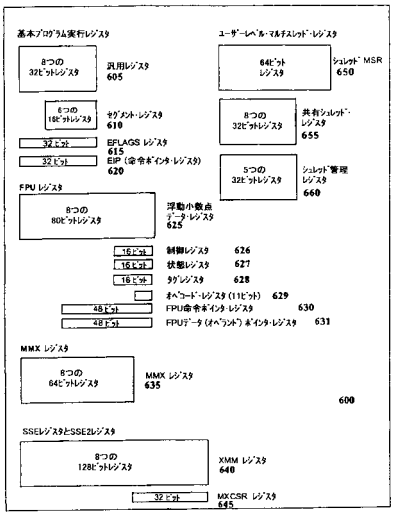
【図３】


300
CPU 310
スレッド 0
320
スレッド 1
321
レジスタ・ファイル
スレッド 0
322
スレッド 1
323
実行ユニット
スレッド 0
324
スレッド 1
325
キャッシュ
キャッシュ
バスインターフェース

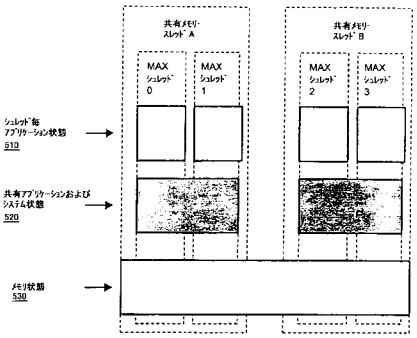
【図４】


400
高性能コア
410
低パワーコア
420
低パワーコア
421
低パワーコア
422
低パワーコア
423
高性能コア
411
低パワーコア
424
低パワーコア
425
低パワーコア
426
低パワーコア
427
共有キャッシュ

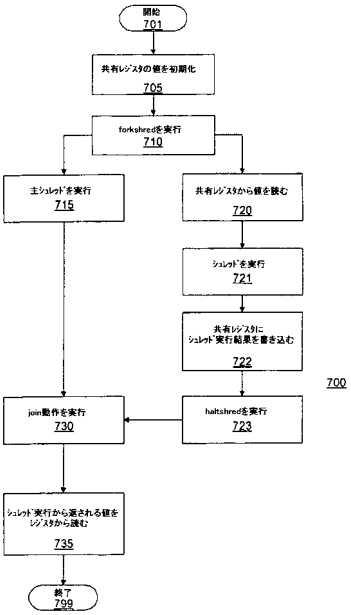
【図５】


基本プログラム実行レジスタ
8つの
32ビットレジスタ
汎用レジスタ
605
6つの
16ビットレジスタ
セグメント・レジスタ
610
32 ビット
EFLAGS レジスタ
615
32 ビット
EIP (命令ポインタ・レジスタ)
620
FPU レジスタ
8つの
80ビットレジスタ
浮動小数点
データ・レジスタ
625
16 ビット
制御レジスタ
626
16 ビット
状態レジスタ
627
16 ビット
タグレジスタ
628
オペコード・レジスタ (11ビット)
629
48 ビット
FPU命令ポインタ・レジスタ
630
48 ビット
FPUデータ (オペランド) ポインタ・レジスタ
631
MMX レジスタ
8つの
64ビットレジスタ
MMX レジスタ
635
SSEレジスタとSSE2レジスタ
8つの
128ビットレジスタ
XMM レジスタ
640
32 ビット
MXCSR レジスタ
645
ユーザーレベル・マルチスレッド・レジスタ
64ビット
レジスタ
シャドウ MSR
650
8つの
32ビットレジスタ
共有シャドウ・
レジスタ
655
5つの
32ビットレジスタ
シャドウ管理
レジスタ
660
600

【図６】


共有メモリ-
ｽﾚｯﾄﾞ A
共有メモリ-
ｽﾚｯﾄﾞ B
MAX
ｽﾚｯﾄﾞ
0
MAX
ｽﾚｯﾄﾞ
1
MAX
ｽﾚｯﾄﾞ
2
MAX
ｽﾚｯﾄﾞ
3
ｽﾚｯﾄﾞ毎
ｱﾌﾟﾘｹｰｼｮﾝ状態
510
共有ｱﾌﾟﾘｹｰｼｮﾝおよび
ｼｽﾃﾑ状態
520
ﾒﾓﾘ状態
530

【図７】


開始
701
共有ﾚｼﾞｽﾀの値を初期化
705
forkshredを実行
710
主ｽﾚｯﾄﾞを実行
715
共有ﾚｼﾞｽﾀから値を読む
720
ｽﾚｯﾄﾞを実行
721
共有ﾚｼﾞｽﾀに
ｽﾚｯﾄﾞ実行結果を書き込む
722
haltshredを実行
723
join動作を実行
730
ｽﾚｯﾄﾞ実行から返される値を
ﾚｼﾞｽﾀから読む
735
終了
799
700

フロントページの続き

(72)発明者 ワン，ホン
アメリカ合衆国 ９４５３８ カリフォルニア州 フリモント サンデール ドライヴ ３９８７７ １０５号
(72)発明者 シェン，ジョン，ピー
アメリカ合衆国 ９５１２１ カリフォルニア州 サンノゼ クウェイル ブラフ プレイス ２３０３
(72)発明者 ワン，ペリー，エイチ
アメリカ合衆国 ９５１３３ カリフォルニア州 サンノゼ キペラッシュ ドライヴ ２９０７
(72)発明者 コリンズ，ジャミソン，ディー
アメリカ合衆国 ９５０３２ カリフォルニア州 ロス ガトス オールド アドビー ウェイ １３４
(72)発明者 ヘルド，ジェイムズ，ピー
アメリカ合衆国 ９７２２９ オレゴン州 ポートランド ノースウェスト １７５ス プレイス ３８９１
(72)発明者 クンドゥ，パルサ
アメリカ合衆国 ９４３０３ カリフォルニア州 パロ アルト エッジウッド ドライヴ １９０６
(72)発明者 レヴィアザン，ラヤ
イスラエル国 ５６５２８ サヴィヨン マガル ストリート １０
(72)発明者 ガイ，ティン－フーク
アメリカ合衆国 ９５１２９ カリフォルニア州 サンノゼ オーク パーク アヴェニュー ５５４４

審査官 漆原 孝治

(56)参考文献 特開２０００－２０７２３３（ＪＰ，Ａ）
米国特許第０５４８５６２６（ＵＳ，Ａ）
笹田 耕一，マルチスレッドアーキテクチャにおけるスレッドライブラリの実装と評価，先進的計算基盤システムシンポジウム ＳＡＣＳＩＳ２００３ 論文集 Symposium on Advanced Computing Systems and Infrastructures，社団法人情報処理学会，２００３年 5月２８日，第2003巻，第9号，ｐ１３－２０
河原 章二，Ｓｉｍｕｌｔａｎｅｏｕｓ Ｍｕｌｔｉｔｈｒｅａｄ（SMT）アーキテクチャの実現方式，情報処理学会論文誌 IPSJ Journal，社団法人情報処理学会，２００２年 4月１５日，第43巻，第4号，p829-843
REDSTONE，J.，et al.，Mini-threads: Increasing TLP on Small-Scale SMP Processors，High-Performance Computer Architecture，2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on，IEEE，２００３年 2月１２日，pp. 19 - 30
佐藤未来子、外６名，マルチスレッドアーキテクチャ向けＯＳ「Ｆｕｔｕｒｅ」におけるプロセス管理，コンピュータシステム・シンポジウム論文集，日本，社団法人情報処理学会，２００３年１２月１１日，Vol. 2001，No. 20，pp. 61 - 70
HANKINS，R. A. et al.，Multiple Instruction Stream Processor，Proceedings of the 33rd annual international symposium on Computer Architecture，IEEE，２００６年 6月２１日

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 9/46
G06F 9/38