マルチスレッドアーキテクチャにおける システムソフトウェアの研究

A study of Systems Software for Multithreaded Architecture

笹田　耕一

Koichi Sasada

(2003 年度入学, 03646109)

指導教官　並木 美太郎 助教授

東京農工大学大学院 工学研究科 情報コミュニケーション工学専攻

2003 年度 修士学位論文

(2004 年 1 月 30 日提出)

目次

図目次

表目次

第 1 章

序論

本論文ではマルチスレッドアーキテクチャプロセッサに適したシステムソフトウェアについて考察する．

本章ではその背景と目的について述べる．

1.1 背景

本節では本研究の背景を述べる．

1.1.1 高性能計算機システムの必要性

近年の計算機技術の急速な発達にともない，コンピュータシステムはさまざまな応用分野で活躍しており，社会基盤のひとつとなっているのはもはや疑いようがない．このような社会的背景から，ますますの高性能計算機が求められている．古くは科学技術計算のために高速な処理が必要されてきた．現在ではインターネットが広く利用されるようになり，そのサービスを提供するサーバなどはより高性能なものが求められている．また，個人が利用する計算機環境も格段の進歩を遂げ，それがより表現力の高いマルチメディアコンテンツなどの計算機上のインターフェースを実現している．これらの表現能力は計算能力により制限されるため，より豊かな表現を実現するためにはより高性能な計算機システムが必要となる．

このような要求に応えるため，高性能計算機システムの実現は必須である．これを行うには，物理的なハードウェアアーキテクチャの開発も必須だが，その上で動作するためのソフトウェアの研究も重要である．これらのソフトウェアを実現するための基盤として，システムソフトウェアが必要である．システムソフトウェアとひとことで言っても，たとえばオペレー

ティングシステム（OS: Operating System）やライブラリなどのミドルウェア，コンパイラやインタプリタなどの言語処理系などがある．高性能な計算機システムを実現するためにはこれらのシステムソフトウェアがハードウェアアーキテクチャを効率よく利用するものでなければならない．

1.1.2 マルチスレッドアーキテクチャプロセッサ

前項で述べたような背景から，プロセッサアーキテクチャも研究が続けられており，近年の速度向上は目覚しいものがある．また，なお研究が続けられている．

従来のプロセッサアーキテクチャは、命令レベルの並列性 (Instruction Level Parallelism, 以下 ILP) に着目して性能向上を目指すアプローチが主流であった。その中で、アウトオブオーダ実行型スーパースカラアーキテクチャは、ハードウェアで動的に一つの命令列からその命令間による依存関係などを解析して並列実行可能性を検出し、それを並列実行することでプロセッサの性能向上に最も成功したアーキテクチャである。さらにこの動的な命令実行のスケジューリングのために、スーパースカラアーキテクチャでは命令ウィンドウを増加させることなどで ILP の抽出を行ってきた。しかし、ILP にだけ注目したアーキテクチャでは、現在性能向上は頭打ちになってきている。これは、プログラム内に存在する ILP はさほど多くないという理由が挙げられる。前述の通り、スーパースカラアーキテクチャの場合は、命令ウィンドウを増強することで ILP をより多く抽出できるが、命令ウィンドウはそのハードウェアの構造上多量に実装すると動作周波数が低下するおそれがある。

この問題を克服するために、ILP とともにスレッドレベル並列性 (Thread Level Parallelism, 以下 TLP) を扱えるアーキテクチャであるマルチスレッドアーキテクチャが提案されている。

マルチスレッドアーキテクチャの一つであるチップ・マルチプロセッサ (Chip Multi Processor , 以下 CMP) は、CPU を一つのチップ内に複数搭載する。また、Simultaneous MultiThreading(以下 SMT) アーキテクチャ [33] [11] [12] [9] は、1 チップで複数のプログラムカウンタ、レジスタコンテキストを持ち、演算器などのハードウェアリソースを共有利用しながら複数の実行命令流を処理することができる。このアーキテクチャについてはすでにいくつかの製品（[15][?][?] など）が発売されており，今後，プロセッサの構成はこの方式が主流になるだろうと目されている．

SMT アーキテクチャは、シングルスレッドアーキテクチャであるスーパースカラアーキテクチャのアプローチでは利用しきれなかった複数の演算器を有効利用し、プロセッサ全体の稼働率を上げようというアプローチである。しかし、演算器など、ハードウェアリソースを共有することでそれらの実行命令流の間で競合が起きてしまう可能性がある。CMP ではそれぞれ

のプロセッサコアでハードウェアリソースを独立して持つため、競合が起こる可能性は無いが、スーパースカラアーキテクチャと同様、利用されない演算器などが存在してしまうという問題がある。

本研究では主に SMT アーキテクチャプロセッサを対象として議論を行う．しかし、本研究で述べる提案は、CMP のようなマルチスレッドアーキテクチャについても適用できると考えられる。この考察については後述する．

マルチスレッドアーキテクチャにおいて，実効命令流を処理する単位を本論文内では**実スレッド**（物理スレッド，AT: Architecture Thread , AThread）と定義する．つまり，マルチスレッドアーキテクチャは複数の計算実体である実スレッドを持ち，それぞれの実スレッドが並列に実行することで TLP をあげ処理性能を向上させることを目標としたアーキテクチャといえる．

1.1.3 マルチスレッドアーキテクチャにおけるシステムソフトウェア

マルチスレッドアーキテクチャを有効に利用するためには、これに適したシステムソフトウェアが必須である。たとえば，オペレーティングシステムでのプロセス管理，スケジューリングを考えたとき，従来の OS では，CPU などの実際に計算を行う実体をカーネルが管理していた．

従来の並列システムと比較してみると，複数のプロセッサをもつ SMP(Symmetric Multi-Processor) 計算機では，それぞれのプロセッサをカーネルが仮想プロセッサとして仮想化し管理する．カーネルはプロセスやカーネルスレッドなどをそれぞれの仮想プロセッサに割り当てる．1 チップに複数の実行実体，つまり実スレッドをもつマルチスレッドアーキテクチャプロセッサを従来の方法で管理しようとすると，1 チップ上に複数のプロセスが動作することになる．このためワーキングセットが広がり、メモリオーバーヘッドにによる性能の低下を招く．

また，その実スレッドにソフトウェアが管理するスレッド（または軽量プロセス）を割り当てることを考えると，その操作にはカーネルへ制御を移さねばならず，効率に問題がある [48]。そのため、高い性能を得るためにはマルチスレッドアーキテクチャに適したシステムソフトウェアが必要となる。

これは，言語処理系にもあてはまり，従来の並列化，最適化手法が適用可能であるかはこの新しいアーキテクチャに対して再考する必要がある．

1.2 本研究の目的

本研究ではマルチスレッドアーキテクチャプロセッサの性能を十分に引き出すことができるシステムソフトウェアについて考察する．その中でもスレッドライブラリに着目して議論する．従来のシステムソフトウェアの枠組みのもとでスレッド処理機構をそのまま利用すると，マルチスレッドアーキテクチャの利点が生かせない．そのため，ハードウェアとシステムソフトウェアの枠組みを再度考察し，マルチスレッドアーキテクチャプロセッサの利点を生かすことができるスレッドライブラリの構築を目指す．

以下，本論文でこれらの点について述べる．2 章ではマルチスレッドアーキテクチャにおけるシステムソフトウェアについて，どのような課題があるかについて述べる．また，従来研究にどのようなものがあるかについて述べる．3 章ではこれらの課題を元に，目標とするシステム全体を設定する．4 章ではその目標とするシステム構成のうち，現状で設計が終了しているものについて，その詳細を示す．とくに，筆者が設計，実装したユーザレベルスレッドライブラリ *MULiTh* と，スレッドライブラリとオペレーティングシステムとの連携について詳述する．5 章でそれらの評価を行い，6 章で結論を述べる．

第 2 章

問題分析と関連研究

マルチスレッドアーキテクチャプロセッサにおけるシステムソフトウェアの構成手法は，従来の並列システム向けに研究され用いられてきた手法と同様に同様に有効なものと，従来の並列アーキテクチャとの相違によりそうでないものがある．また，新しいプロセッサアーキテクチャであるため，構成手法も新たな手法を適用できる可能性がある．

本章ではまずマルチスレッドアーキテクチャの性質を述べる．そして，その性質が従来のシステムソフトウェアにおいてどのような課題があるか，オペレーティングシステムやライブラリ，言語処理系の視点から述べる．また，これらの課題に対する既存の関連研究についても示す．

2.1 マルチスレッドアーキテクチャプロセッサの性質

本節はマルチスレッドアーキテクチャプロセッサの性質について従来の並列処理用システムと比較して特徴的な点について述べる．本節で述べる性質は比較対照として共有メモリ型マルチプロセッサシステムで，主に対称型マルチプロセッサ（SMP: symmetric multi-processor）を念頭に述べる．

2.1.1 計算資源の共有と競合

並列処理システムを構築するとき，計算資源がどのように配置されるかが性能上非常に重要である．ここでいう計算資源は主メモリ，キャッシュメモリ，プロセッサ内の演算器や各種バッファ，バス，入出力装置など多岐にわたる．

計算機システムでは，それぞれの資源を計算実体同士で共有するか，共有しないかという点

が重要である．一般に，ある資源を共有するとその資源の効率的な利用が可能になり，その資源を用いた通信は速くなるが，規模を大きくするのが困難になる．また，資源を共有するためどの計算実体がそれを利用するかの決定が問題になる．これを資源の競合という．競合を解決するためにはどの計算実体がその資源を利用するかを決定するためのスケジューリングが必要になる．たとえば，一般的な OS のプロセススケジューリングは，プロセスという仮想的な計算実体を物理的な計算実体へと割り付ける資源管理スケジューリングということができる．

共有しない場合，その資源の規模を大きくすることが可能になるが，たとえばそれぞれの資源での一貫性を保たなければならない処理がある場合，そのための機構が必要になる．一貫性はメモリなどでよく取り上げられる問題である．また，資源を共有しない場合，ある計算実体用の資源は利用されていないが，ある計算実体ではその資源が枯渇するようなことも考えられ，効率的な利用がされない可能性がある．

グリッド・コンピューティング (GRID) やクラスタコンピューティングのような（大規模）分散システムではこれらの資源はすべて計算実体ごとに確保する．これはそれぞれの計算実体の作成と確保を容易にし，大規模なシステムを構築することを可能にする．しかし，計算実体同士では粗結合となり，通信に時間がかかる．

入出力装置などを共有する場合，たとえば一台の計算機に複数の計算実体を持たせるような並列システムでは，たとえば CPU を複数個持たせるマルチプロセッサのアプローチの場合がある．このときでも主メモリを共有するかが問題となる．また，主メモリを共有しても，各プロセッサから物理的なメモリへのアクセス速度が異なるアーキテクチャ（NUMA: Non-Uniform Memory Access、Non-Uniform Memory Architecture）もあり，システムソフトウェアはこれを考慮したものが用いられなければならない．たとえば，OS はプロセスの割り当てとページングをアクセスが速い部分に行えばシステムの性能は向上する．これがすべて均一なシステム（UMA: Uniform Memory Access、Uniform Memory Architecture）では，これらを考慮する必要はない．SMP システムなどはこれにあたる．

SMP システムでは CPU 内部以外の資源は一般に共有される．さらに資源を共有するには，その CPU 内部の資源について考える必要がある．つまり，これがマルチスレッドアーキテクチャプロセッサである．マルチスレッドアーキテクチャの場合，ひとつの CPU に複数の計算実体が含まれている．そのため，CPU 資源を共有することが可能になる．

チップマルチプロセッサ（CMP）はほとんど SMP 構成のシステムを 1 チップに納めたような構成になっている．この構成の利点はいままでの SMP のシステム構成と同様に考えることができるといことである．ここで共有できるのはキャッシュメモリや CPU に接続されているキャッシュメモリなどである．

SMT アーキテクチャでは各計算実体が演算器なども共有する．計算実体として最低限必要

なコンテキスト保存用のレジスタセットなどを個別にもち，他の部分については共有するというアプローチである．

もちろん，各アーキテクチャがどの計算資源をどのくらい共有するかについては議論の余地がある．たとえば SMT プロセッサにおいてもパイプラインのどの部分を共有するか，しないかは大きな技術的課題である．また，キャッシュメモリについても一般的に 1 次キャッシュ，2 次キャッシュ，3 次キャッシュなどあるが，どのキャッシュメモリから共有するかなどについても考慮すべき問題である．共有する度合いにおいても，すべての計算実体があるひとつの計算資源を共有するのか，ある程度グループ分けして共有するのかなどの課題がある．

ただ，一般にマルチスレッドアーキテクチャは他の並列計算機のモデルよりも密結合であり，各種資源を共有するというのは確かな点である．本項では演算器の共有とキャッシュメモリの共有についてもう少し掘り下げて特徴を述べる．

演算器の共有

スーパスカラプロセッサアーキテクチャは，命令列中の命令レベル並列性（ILP）に注目し，並列実行可能な命令を並列に実行しようというアプローチである．並列に実行するためには演算するために必要な資源が複数必要になる．その一つが演算器である．演算器は単純な算術演算（足し算や引き算），論理演算などを行う Simple ALU（Arithmetic and Logical Unit）や複雑な計算を行う Complex ALU，分岐命令などを扱う Branch Unit，メモリのロード，ストアを行うロードストアユニット，浮動小数点演算を行う FPU（Floating Point Unit）などがある．とくに，Simple ALU などは 1 チップ上に複数載せることが容易であるため，近年のプロセッサでは複数搭載している場合が多い（たとえば，Intel Pentium4 プロセッサの場合，Simple ALU にあたるものが 2 個，またそれぞれ 1 クロックで 2 命令発行可能 [6]）．

しかし，ILP が思うほど抽出できない場合，これらの演算器の利用率が下がる可能性がある．事実，特別なコンパイラの最適化などをしない多くのプログラムではたいした ILP は出ない．そのため，本来の CPU の性能を十分に発揮できているとは言いがたい．これは，命令レベル並列性の抽出自体が難しいものであるといえる．プロセッサレベルでの命令列の解析による ILP の抽出の精度をあげるためには，その解析のためのハードウェアの増設が必要となり，困難である．そのため，近年ではプロセッサの性能を引き出すためコンパイラなどの役割が重要となっている．

ここで，他の並列性について目をつけたのがマルチスレッドアーキテクチャである．一つの命令列から並列性抽出が困難ならば，複数の命令列を利用して並列度（TLP: スレッドレベル並列性）を抽出しようというアプローチである．これにより，演算器の利用率を向上すること

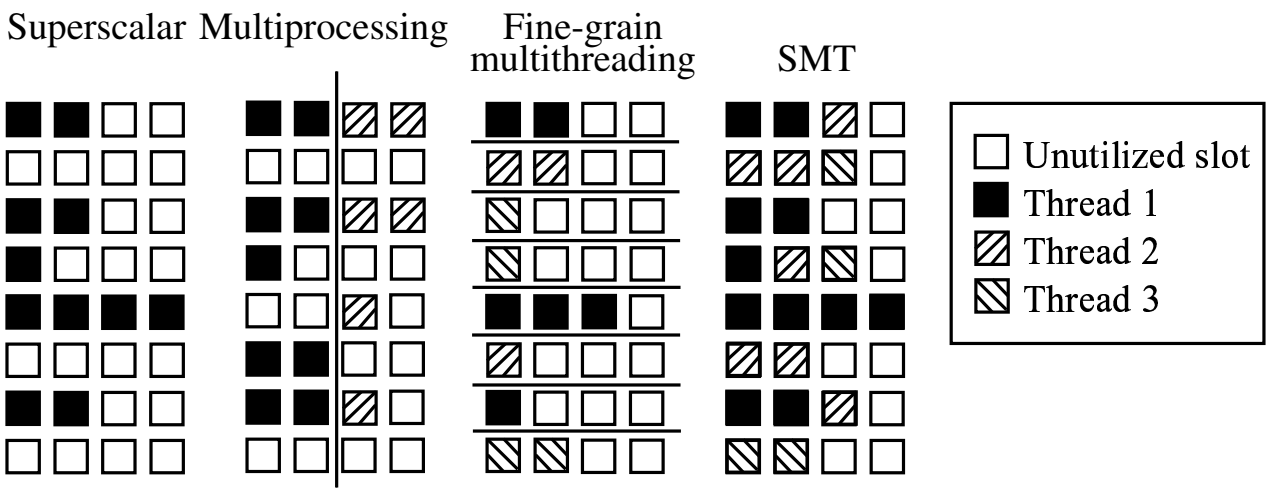

図 2.1: 各種アーキテクチャの命令発行の様子

ができる．

CMP プロセッサのような場合，演算器を共有しないため，この問題は解決しない場合があるが，そもそもそれぞれの計算実体がもつ演算器の個数を制限することでスーパスカラアーキテクチャで問題となっていた利用率の低下が解決する可能性がある．しかし，複数の命令列が供給されない場合，計算実体自体が利用されず，効率が落ちる可能性がある．

SMT プロセッサのような場合，すべての計算実体が演算器を共有するので，一番無駄がなく効率的に動作することができる [19]．また，命令列が十分に供給されない場合でも，計算実体それぞれがスーパスカラプロセッサと同様に ILP を生かすよう実行するようにすれば，スーパスカラプロセッサと同様のパフォーマンスで実行することができる．

これらを図示したのが図 2.1 である．これは演算器利用率（正確には命令発行）を示している．左はスーパスカラプロセッサの演算器の利用の様子を示しており，ILP が十分でない場合には演算器の利用率が低いことがわかる．中央の左はマルチプロセッサ構成，およびチップマルチプロセッサを示している．スーパスカラプロセッサよりは演算器の利用率は高くなっているが，一つの命令列の ILP に強く依存するような形になっている．図の中央右は同時実行しないマルチスレッドアーキテクチャを示しており，1 サイクル中の演算器利用率は ILP によることがわかる．図の右は SMT プロセッサを示しており，演算器の利用率がもっとも高いことがわかる．このように，複数の計算実体による演算器の共有は，TLP を生かすことで有効に演算器を利用できていることがわかる．

しかし，共有することによるデメリットも発生する．計算資源を共有するため，その資源に関する競合が発生する．そのため，十分な性能が発揮されない場合がある．

たとえば，命令列 A がある演算器を利用しているところで命令列 B もそれが必要になったとするとここで競合が発生する．たとえば Simple ALU のような高速に実行できるものであればこれはあまり問題にならないが，Compex ALU や FPU のような実行に時間がかかるものの場合，この競合により後の命令が実行できなくなる可能性があり，たとえば命令列 A のみで実行していたほうが全体のスループットが向上する可能性がある．これは，ILP が十分高い命令列や，コンパイラによって静的に解析し，最適なスケジューリングが行われた命令列において，とくにこの問題が起こる可能性がある．

また，演算器資源を共有するので，一つの命令列の総実行時間は，それが他の命令列と並列に実行していなかった場合，つまり従来の逐次実行と同様に行っていた場合よりも長くなるのが一般的である．つまり，命令列 A, B があったとき，A と B の総実行時間は A,B のみで実行した場合，それぞれ 2 であったのが，A と B で並列実行した場合，それぞれが 3 になったとする．しかし，A, B 両方走らせた合計時間も 3 であれば，A と B ですべての処理と考えたとき，全体としては短くなっている．つまり，全体のスループットは向上するということである．これは，リアルタイムシステムなどに影響する可能性がある．計算能力のみを求める科学技術計算などでは，これは問題にならないが，組み込みシステムや実時間処理を要求するインタラクティブ性を重視するようなマルチメディア処理などでは問題になる場合がある．

キャッシュメモリの共有

近年の CPU の驚異的な高性能化とは裏腹にメモリの高性能化が遅れており，両者の速度的なギャップは年々大きくなっている．これは，大きな主メモリを要求するソフトウェアに対応するために大容量のメモリを搭載することが一般的になっているが，大容量メモリをすべて高速なメモリとするとコスト的に現実的ではなくなってしまうからである．このような背景からキャッシュメモリが性能に重要な要因となっている．

キャッシュメモリは小容量ながらアクセス速度が速いメモリを CPU 内部，もしくは外部におき，主メモリとのメモリのやり取りをキャッシュ（chache）してメモリアクセスの向上を目的とするものである．一つのキャッシュだけでなく，それを何段か重ねて，1 次キャッシュ，2 次キャッシュ，...，n 次キャッシュを搭載する場合がある．キャッシュメモリをチップ内に置くかは，プロセッサアーキテクチャについて大きな課題であるが，一般的には 1 次キャッシュはプロセッサに搭載し，2 次キャッシュ移行はケースバイケースであることが多いようである．

プロセッサ内部にキャッシュメモリを搭載すれば，そのキャッシュメモリとの通信は高速になるが，CPU の面積が大きくなってしまうのでチップの構成が難しくなる．このあたりのトレードオフは重要な課題である．

さて，プロセッサ内部のキャッシュを考えた場合，プロセッサ内部で複数の計算実体を扱うマルチスレッドアーキテクチャではこれを共有することができる．これについての長所と短所について本項では論ずる．

キャッシュをめぐる問題としては，並列システムではキャッシュメモリの一貫性（コヒーレント）制御の問題が有名である．これは，それぞれのキャッシュメモリで同一のデータをキャッシュしたとき，そのデータをある一つの計算実体が変更した場合，どのようにその変更を処理するかという問題である．そのためのアルゴリズムも実に様々な研究がある [22]．しかし，どの手法も一般になんらかの装置やソフトウェアによるオーバヘッドがかかることが必須である．

キャッシュメモリを複数の計算実体で共有したとき，このコヒーレント制御の問題は解決する．すなわち，それらのオーバヘッドが必要なくなるということである．これは，キャッシュメモリを共有するようなプログラムの実行性能が向上することを意味する．たとえば，他の計算実体と通信を行う必要があるとき，主メモリへ書き込むことでキャッシュメモリにもその情報が残りそのキャッシュメモリ上の情報を他の計算実体が安価に読み込むことができる．

これは，たとえばロック変数をメモリ取るようなソフトウェアで有効に働く．また，一つのデータを複数のスレッドが並列に処理するようなプログラムの場合もこれは有効に働く．また，パイプライン型のプログラムでは，情報を他の計算実体に伝達することが必須になるが，これもキャッシュメモリを通じての通信で容易に行うことができる．これらの詳細は次項でまた述べる．

キャッシュメモリを共有した場合のデメリットは，これも競合である．

仮想メモリシステムでは，使われなくなったメモリ領域を記憶装置にスワップアウトする．スワップアウトした領域が必要になれば，それをまたメモリへ読み込むスワップインが行われるが，これが頻繁に繰り返されるとシステムの性能が急激に落ちる可能性がある．これはスラッシングとして知られている．これと同様の問題がキャッシュメモリを共有した場合に起こる．

つまり，ある計算実体がキャッシュした領域を，ほかの計算実体が別の場所をキャッシュするために追い出してしまう．このとき，キャッシュを共有していなければこのような問題は発生せず，より高速に処理を終了することができる．この問題は，それぞれの計算実体のワーキングセットがそれぞれ大きく，そのメモリ領域の共有部分が少ないときに顕著に生じる．

しかし，キャッシュを共有しないようにすると，それぞれの計算実体に割り当てるキャッシュメモリの容量は少なくなる．計算実体の数が n 個あり，キャッシュメモリのサイズが S で，それぞれ均等に割り当てるとしたら各計算実体に割り当てられるキャッシュメモリの大きさは S/n となる．これは，ワーキングセットの大きなプログラムを実行するときに問題と

なる．

また，ある計算実体はキャッシュメモリをあまり利用せず，他の計算実体ではキャッシュメモリが枯渇しているような場合，キャッシュメモリが効率的に利用されているとはいえない．これは，演算器の共有と同様の問題である．

メモリアクセス対象のメモリがキャッシュメモリにのっていなかったことによる命令実行遅延であるキャッシュミスペナルティは非常に大きく，最悪 100 サイクルほどかかってしまう場合がある．これは，演算器の競合による速度低下に比べて非常に影響が大きい．

まとめると次のようになる．

演算器の共有による長所は，主に演算器の利用率の向上による速度向上と言える．また，演算器の競合による悪影響は最適な命令スケジューリングが崩れることによる悪影響であるが，通常のシングルプロセッサでの実行で競合を行われていたときの動作速度とあまり変わらないことが期待できる．

キャッシュメモリの共有により通信性能が向上する可能性があるが，キャッシュメモリの競合（とくに，スラッシングのような減少）が起こった場合のシステムの性能低下は大きな問題となる．

つまり，演算器の共有は処理速度向上のための工夫，キャッシュの共有は処理速度を落とさないための工夫となり，またそのための機構を考察する必要がある．

2.1.2 通信

並列プログラムでは，各計算実体が情報を通信する必要がある．たとえば，分割統治型の並列プログラムの場合，計算実体を生成し，それに初期値を渡す必要があり，またその計算結果を作成側に返す必要がある．Do/All 型のような同一処理を繰り返すような場合でもそのイテレーションのインデックス情報が必要になる場合がある．パイプライン型の処理はそれ自体が次々に他の計算実態に情報を伝達していく計算モデルである．

また，処理によってはクリティカルセクションなどを設けなければならず，排他制御や同期のために，他の実行実態の情報を知る必要がある．同期は，すべての計算が終了したことを待つ場合などのバリア同期などでよく必要とされる．これが効率よく行われないと，処理自体は効率よく並列化できたとしても，その同期処理のオーバヘッドにより逐次実行時よりも速度が遅くなってしまうことがある．また，そもそも計算実体の管理もそれらの情報のやり取りを必要とする．

このように情報の通信は並列処理の性能に重要な要因である．本項ではマルチスレッドアーキテクチャプロセッサと従来の並列システムと比較してどの点が異なっているかについて述

べる．

データの通信

計算の初期値や計算結果などを異なる計算実体へ受け渡す方法について考えてみる．受け渡す値の種類はもちろんプログラムによっても違うが，数ワードのものから数 MB ほどの大容量のものまでさまざまである．大規模分散処理システムではこれらのデータ受け渡しには非常にコストがかかり，特殊な処理が必要になるが，共有メモリシステムの場合，ただ共有メモリに対してメモリを書き込むだけでよく，特殊な処理がいらない．

マルチスレッドアーキテクチャプロセッサの場合，前述したようにキャッシュメモリを共有している場合があるのでさらに高速な情報伝達が可能になる．SMP システムでは，共有メモリに書き込んだだけではそれぞれに備えられているキャッシュメモリのみに書き込まれる場合があり，明確な一貫性制御などが必要となる．マルチスレッドアーキテクチャの場合，これがいらない．

また，単純にキャッシュを共有しているため，主メモリへのアクセスがいらなくなり，メモリバスなどにアクセスする必要がない．その点でも十分高速であることがわかる．

排他制御や同期などを行うためにはメモリに関してロック機構を設ける必要がある．たとえば IA32 の xchg 命令 [7] や MIPS アーキテクチャの LL/SC 命令 [29] などは，これを行うために必要なアトミックなメモリアクセス命令であるが，複数の CPU と一貫性をとりこれを行うにはコストがかかるが，チップ内での計算実体間でこれを行うのはそれにくらべて安価であることがわかる．

つまり，キャッシュメモリを共有していれば，効率のいいデータ通信が行え，アトミックなメモリアクセス命令も比較的容易に実装できる．

計算実体の管理

並列に計算実体を動作させるには，その計算実体の制御が必要になる．SMP システムではこの制御を他のプロセッサに対する割り込みなどで実現していた [7] が，マルチスレッドアーキテクチャの場合，並列実行単位が同一 CPU 内にあるのでたとえば 1 命令でその制御が可能になる．

たとえば計算実体の割り当てや解除，一時停止やその解除などが 1 命令，1 サイクルで行うようになると，これらの処理をより有効に生かしたシステムソフトウェアが考えられる．

また，チップ内での計算実体の管理をプロセッサが行うことができ，たとえば現在何個の計

算実体が実際に稼動しており，新しい計算実体が割り当て可能であるかどうかをプロセッサが管理することができる．これは従来システムソフトウェアが行っていたことである．

このような処理が安価にできるのはマルチスレッドアーキテクチャならではの特徴である．

2.2 オペレーティングシステムやライブラリの課題

本節ではマルチスレッドアーキテクチャにおけるオペレーティングシステムやライブラリの課題について述べる．また，ライブラリは主にスレッドライブラリについて述べる．

2.2.1 プロセス管理とスレッド管理

ここでいうプロセスとは UNIX 流のプロセスのことであり，スレッドも同様である．これらの詳細は [42] に詳しい．ここではそのなかから基本的な部分を述べておく．

UNIX に代表される古典的な OS では、一つのプロセスは一つの仮想アドレス空間と一つの制御フロー（狭義のコンテキストで、プログラムカウンタ、レジスタの内容およびスタックなどをさし、仮想アドレス空間とファイル資源は含まない）から構成されている。

しかし、複数の処理がアドレス空間を共有し、同期をとりながら強調的に並行、並列動作させたいという要求が多くなっている。そこで、制御フローだけを独立して抽象化しようという考えが生じてくる。この制御フローが” スレッド (thread)” である。スレッドは生成、消滅、同期のオーバーヘッドが比較的小さいので軽量プロセス (lightwaight process) と呼ばれることもある。本論文内でスレッドという言葉を用いる場合はこの定義を示しているものとする。つまり、古典的な UNIX のプロセスは、一つのスレッドを所有している、と言うことができる。

プロセスとスレッドおのおのが保持し管理する情報は、厳密には OS によって異なるが、UNIX などのシステムでは表 2.1 のようになる。

つまり、スレッドはプロセス中のすべてのメモリを共有し、それぞれのスレッドはプログラムコードが指示するように、プロセスのメモリを読み書きしたりすることができ、あるスレッドがメモリに書き込むと、別のスレッドがその結果を読み込むことができる。

スレッドの概念のない古典的な UNIX 系 OS では、メモリを共有するような複数の制御フローや、複数の OS の機能をアクティブにするには、プロセス自体を複数生成し、パイプやソケットのようなプロセス間通信を利用して互いに通信しあうことにより実現してきた。だが、それらの機能を用いるのはその準備やデータの受け渡しで比較的大きなオーバーヘッドがあることがわかっている。

表 2.1: スレッドとプロセスの基本的な情報

スレッド	プロセス
プログラムカウンタ	仮想アドレス空間
スタック	大域変数
汎用レジスタの内容	開いたファイル
スレッドの属性	タイマ
	シグナル
	セマフォ
	アカウント情報

並列システムで問題になるのが，このプロセスとスレッドを誰がどのように管理するかということである．従来のシステムソフトウェアでは，CPU などの計算実体をカーネルが管理し，それにプロセスを割り当てるようにスケジューリングを行っていた．つまり，プロセスは仮想プロセッサとして扱われていたのである．

では，スレッドの管理はどのように行えばいいだろうか．カーネルがプロセスの特異なものとして管理するか，それともまた別の管理手法を用いるかである．この問題はスレッドという概念が考え出されてから長く議論されてきた問題である．その結果，スレッドを管理する手法は，おおざっぱに言って，ユーザレベルで管理する方法とカーネルが管理する方法，またその折衷案とも言える 2 レベルスレッドモデルが提案されてきており，それぞれ特徴がある．本項では以下，それぞれについて議論する．

カーネルレベルスレッド

カーネルが物理プロセッサを仮想化して提供する仮想プロセッサ（カーネルスレッド）としてスレッドを提供する。生成したスレッドはカーネルスレッドに 1 対 1 にマッピングされて実行されるモデルである（図 2.2)。1 対 1 モデルとも呼ばれる。スレッドのスケジューリングは、カーネルのスケジューラが行う。

カーネルスレッドを提供しているオペレーティングシステムは、Mach[32] などが有名であるが、Linux での Linux Thread[34] や、次期バージョンの Linux 用スレッドライブラリといわれている NPTL[8] などもこのモデルである。

本モデルは UNIX のプロセスの自然な改良といえる。この場合、スレッドの制御、管理は

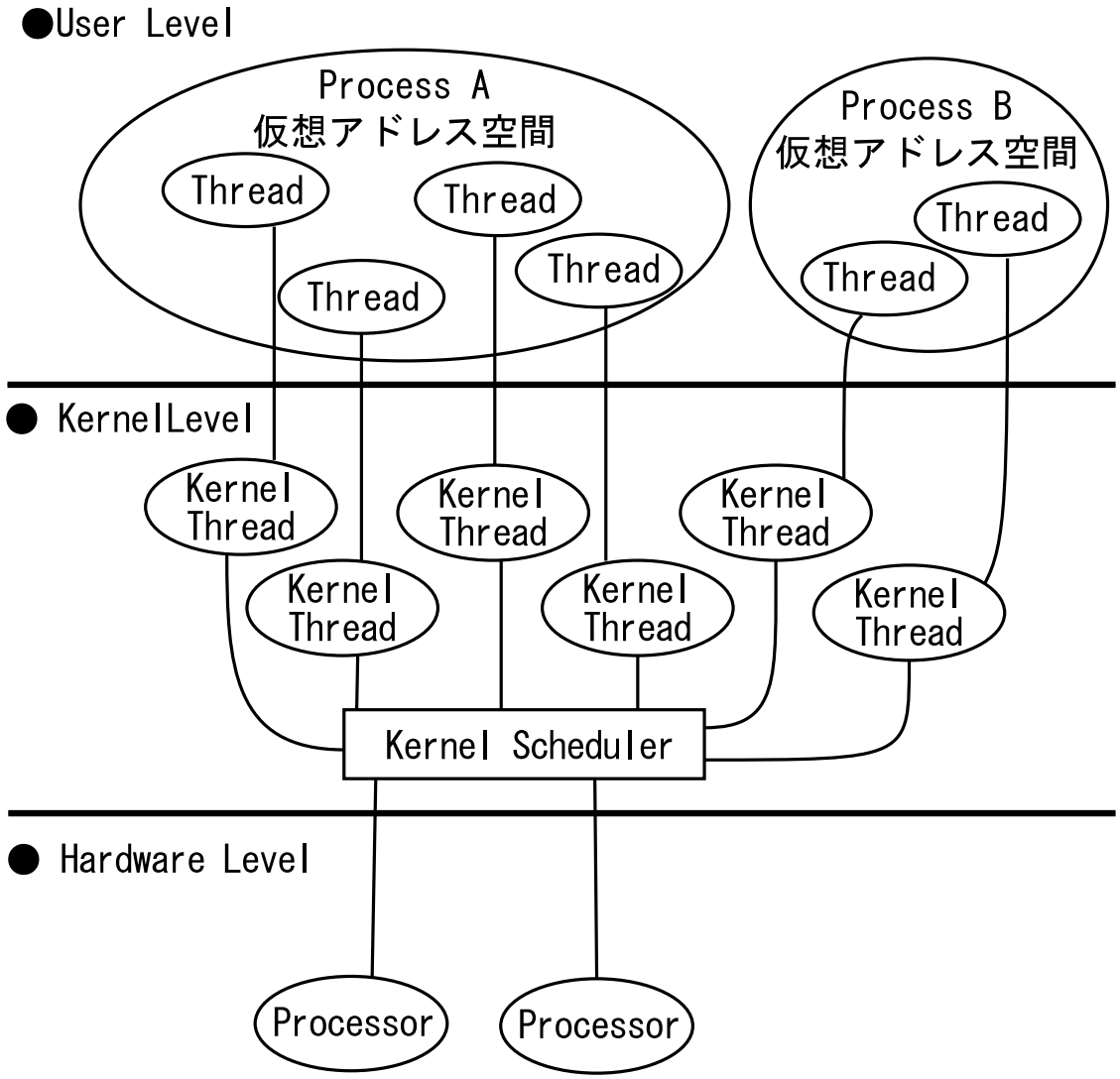

図 2.2: カーネルレベルスレッドモデル

OS によって実現される。本モデルはスレッドがカーネルスレッドとして実行されるので、後述するユーザレベルスレッドモデルに比べてスレッドのカーネル内での動きがユーザ（空間）に見えやすいという利点がある。また、スケジューラが一つしかないのでスケジューリングのオーバーヘッドを減らすことができると考えられる。反面、次のような欠点があることがわかってきた。

まず、スレッドの制御に高い負荷がかかる。これらの生成、削除などのスレッド制御がシステムコールを発行しなければならないためである。システムコールの発行は仮想アドレス空間をユーザ空間からカーネル空間に切り換え、ユーザ空間のスタックに詰められているシステム

コールの引き数をカーネル空間にコピーしなければならない．また，プロセッサのモードも変更しなければならず，制御の擾乱となる．カーネルレベルスレッドモデルでは，これらのオーバーヘッドが発生し、高い性能を引き出すことができない．

また、柔軟なスレッドプログラミングが行えない。スレッドモデルはカーネルが規定することになるので、ユーザはカーネルが規定したスレッドモデルにあわせてプログラミングをしなければならなくなる。また、スレッドスケジューラはカーネルが提供する単一のものとなるため、異なるユーザプログラムに対しても同一のスケジューリングポリシを適用することになり、おのおののユーザプログラムに適したスケジューリングポリシを実現しにくい。また、カーネルスレッドでより多くのスレッドモデルを包含しようと思えば、カーネルが種々の機能を提供しておく必要がある。すなわち、あるプログラミングモデルでは必要ない機能でも、他のプログラミングモデルにとって必要な機能は提供しておかなくてはならない。

ただし，並列処理プログラムを作成するとき，ある程度粗粒度なもの，たとえば各プロセスは粗粒度並列性を有するということができるが，これに類するようなスレッドを作成する場合，これらのオーバヘッドなどは気にならないこともある．また，細粒度並列性をもつプログラムを書くのは（問題が本質的にそれを有していない限り）一般に難しいため，このモデルで十分な場合がある．前述したとおり，このモデルはシンプルであるため近年ではこのモデルが支配的であると言える．

ユーザレベルスレッド

ユーザレベルスレッドモデルは、ユーザ空間でスレッドの生成、消滅およびコンテキストスイッチなどのスレッド管理を行うものである（図 2.3)。プロセス生成時に一つカーネルスレッドができ、そのカーネルスレッドに対して複数のスレッドを割り当てて実行するため、多対 1 スレッドモデル、または M のスレッドを 1 つのカーネルスレッドにマッピングするモデルとして、M 対 1 モデルともいう。

ユーザ空間内の実行時ルーチンでユーザスレッドをスケジューリングできる。すなわち、ユーザスレッドのスケジューリングポリシをそのプログラムに適したポリシにユーザが定義できるという観点からは、効率的なスレッド制御が行える可能性がある。実行コンテキストスイッチをユーザ空間だけで行えるかどうかはプロセッサアーキテクチャに依存するが、殆どのプロセッサではユーザ空間だけで可能である。このスレッドモデルでの処理機構は一般的に、プログラムにリンクして利用するようなライブラリとして提供される。

実装方法としては、多田ら [39] の `setjmp/longjmp` を用いた方法が移植性が高いとして多く実装されている。安部らの研究 [41] では、シグナルを用いた移植性の高いプリエンプティ

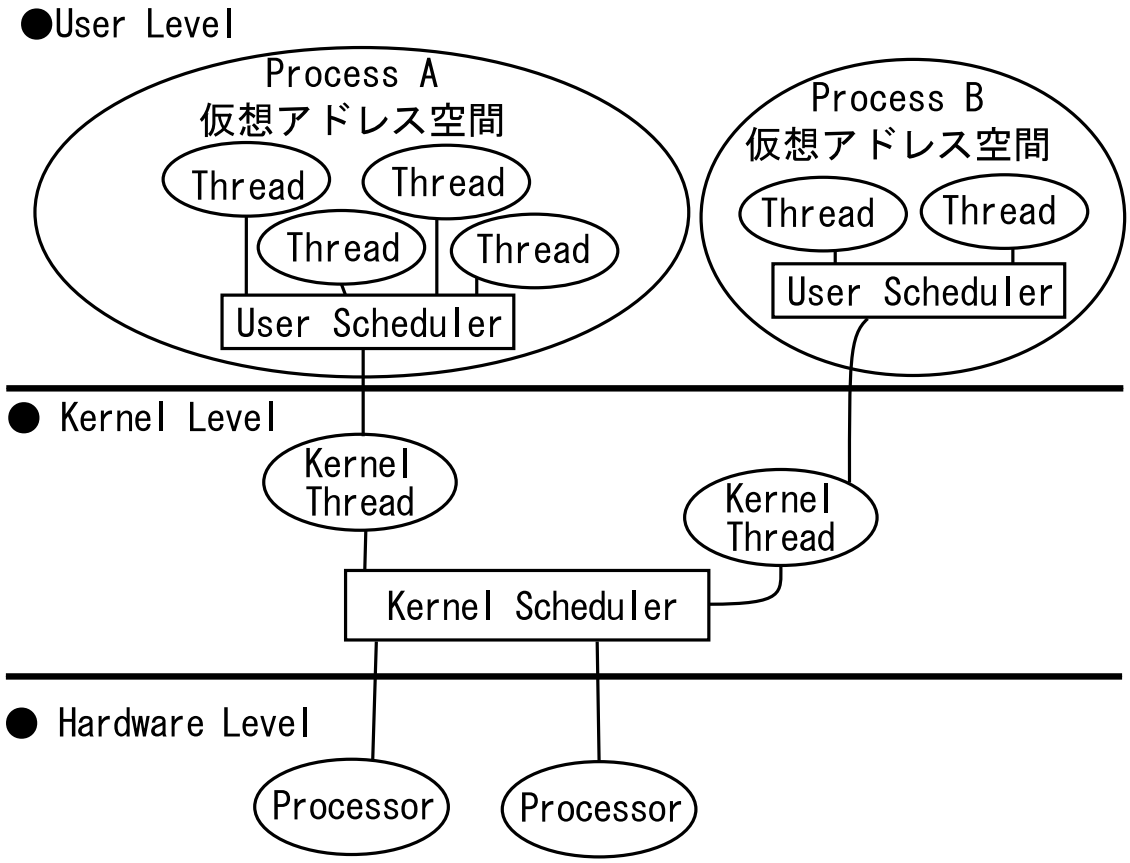

図 2.3: ユーザレベルスレッドモデル

ブなコンテキスト切り換えを実現している。

長所としてはカーネルレベルスレッドモデルのようにユーザモードからカーネルモードへの空間切り換えを必要とせず、関数呼び出し程度の軽さでスレッド制御が実現できることである。

短所として、ユーザ空間から仮想プロセッサの物理プロセッサへのマッピング状況を知ることができない点である。特に、仮想プロセッサがブロックされたことがユーザ空間から見えないことが問題となる。仮想プロセッサがブロックされる契機として、次のような場合がある。

1. カーネル空間実行中のブロック（I/O リクエストなど）
2. ユーザ空間実行中のブロック（ページフォールトなど）

この問題は、一つのスレッドがブロックすると、他のスレッドへ処理が移らなくなってしまうという点にある。待ち状態のスレッドがある場合、そのスレッドへ処理を渡すことができない。

また、この方法では一般的にカーネルの保護の元にある複数プロセッサなどの利用ができないという問題点もある。これは、カーネルスレッド一つに対してユーザレベルのスレッドを複数マッピングするため、各スレッドの並列実行ができない。

2 レベルスレッド

それぞれ、カーネルレベルスレッドモデルとユーザレベルスレッドモデル、どちらの方法も一長一短があった。それらの長所を維持するようにスケジューリングモデルを作ったのが 2 レベルスレッドモデルである。これは、N 個のカーネルスレッドで M 個のユーザレベルで作成した実際のスレッドをスケジューリングする方法で、M 対 N モデルとも呼ばれる（図 2.4)。スレッドの制御はユーザレベルで行うため，制御は軽量に行うことができる.

これは、一般に $N \leq M$ であることが多く、M 個のスレッドを N 個のカーネルスレッドに割り当て、多重化する。すなわち、ユーザレベルのスレッド制御機構（一般的にはライブラリとして提供される）は、スレッドをカーネルレベルスレッドにスケジューリングし、そして今度は OS がカーネルスレッドを利用可能な物理プロセッサにスケジュールする。この意味では OS の管理するカーネルスレッドを仮想プロセッサとみなすことができ、スレッドライブラリをマルチプロセッサ OS の縮図とみなすことができる。

この機構は一般的に、OS の管理するカーネルスレッドが存在し、それをユーザが複数管理できるという条件の下でライブラリの形で提供される。実際の実装は、メモリマップドファイルを用いた小熊らの研究 [49] や、これも次期バージョンの Linux に採用候補であるといわれている IBM での NGPT[13] 、安部らの I/O ブロッキング回避たのめに他プロセスを生成する研究 [40] などがある。

この機構により、OS の管理する複数のプロセッサを利用することができ、またスレッド制御をユーザレベルで行うことにより、効率よく行うことができる。ただし、スレッド割り当ての対象を物理プロセッサではなくカーネルが提供するカーネルスレッドとしているため、実際にそのカーネルプロセスがどのようにスケジューリングされているのか、スレッドライブラリは知ることができず、効果的なスケジューリングをすることができない。また、ユーザレベルスレッドライブラリの項で述べた、カーネル内でのスレッドブロッキングの問題は同様に発生する。

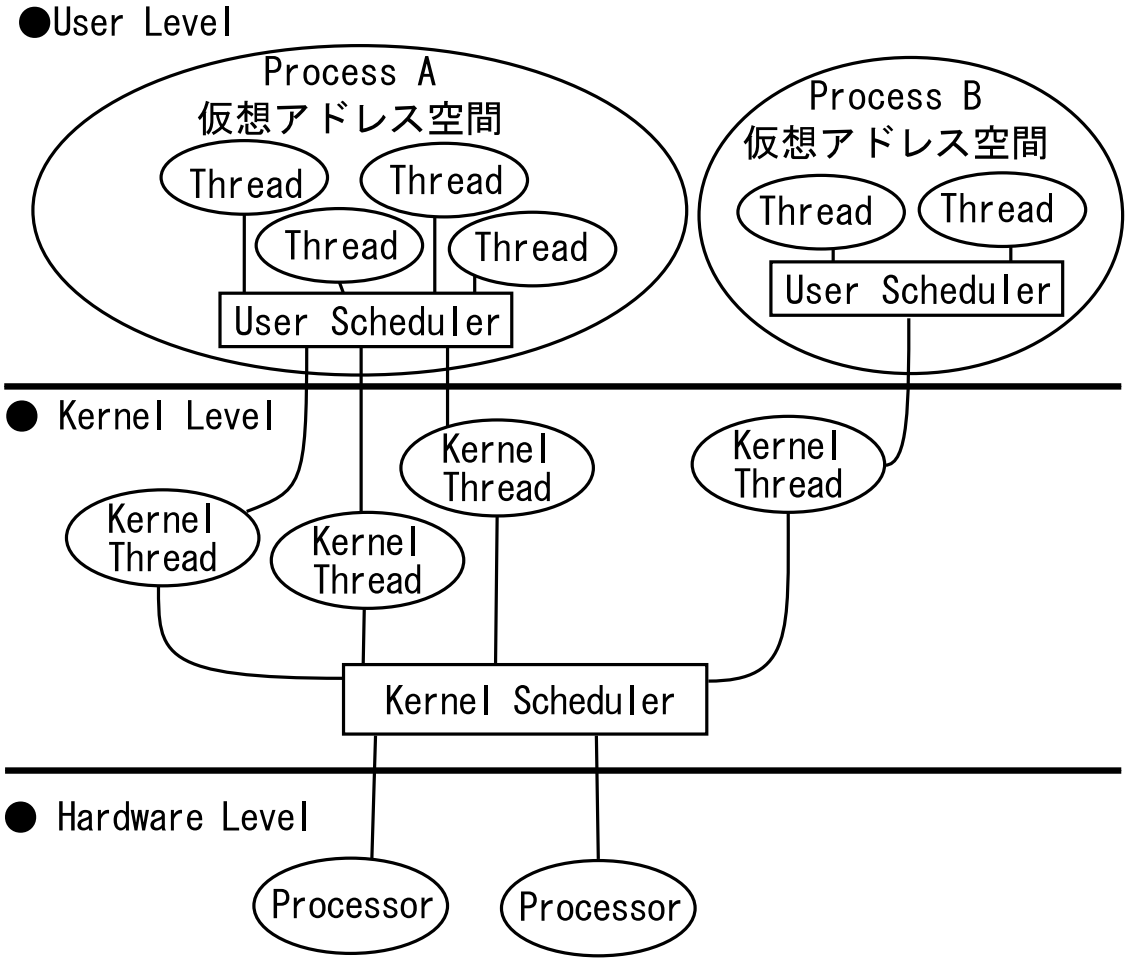

図 2.4: 2 レベルスレッドモデル

協調型ユーザレベルスレッドモデル

前述したように、ユーザレベルでスレッドを管理するモデルではいくつかの問題があり、それらの問題の多くはカーネル側の状態をユーザレベルでは知ることができない、ということが原因である。すなわち、システムコールという形で新たな情報の流れがユーザ空間からカーネル空間への 1 方向しかない点である。そこで、そのような情報伝達経路を持たせる研究がある。これは、カーネルレベルスレッドモデルがもつ機能性とユーザレベルスレッドモデルが持つ高機能性と柔軟性の両方の長所の提供を目的としたものである。

Washngton 大学では、スケジューラアクティベーション (Scheduler Activation) と呼ばれ

る機構を提案している [4]。これは，カーネル空間からユーザ空間の情報伝達にアップコール (Upcall) 機構を用いている。また、本機構を提供する仮想プロセッサもスケジューラアクティベーションと呼んでいる。スケジューラアクティベーションは次の三つの働きをする。

1. ユーザスレッドを実行するコンテキストを提供する
2. カーネル内で生じたイベントをユーザ空間に知らせる
3. カーネル空間内でユーザスレッドがブロックされたときのコンテキストの保存領域を提供する

従来のユーザスレッドモデルとスケジューラアクティベーションの違いは、カーネルスレッドがブロックされたときどのように動作するかという点である．ユーザレベルスレッドモデルでは、仮想プロセッサがブロックされると、その上で走っているユーザスレッドも自動的にブロックされ、仮想プロセッサがブロックされたことがユーザに通知されない。したがってユーザスレッドのブロックが、ユーザ空間から見えないことになる。また、ブロックされたかそうプロセッサを再開させる際もユーザ空間に通知されない。

一方スケジューラアクティベーションではブロックされるとカーネルは新たなカーネルスレッドを生成し、この生成されたカーネルスレッドがユーザレベルへブロックされたことを通知し、処理をユーザ空間のスケジューラにゆだねている。

スケジューラアクティベーションと似たスレッドモデルが Rochester 大学で提案されており、並列オペレーティングシステム Psyche[25] 上に実現されている [20]。彼らは提案したスレッドモデルをファーストクラスユーザスレッドと呼んで、今までのユーザレベルスレッドモデルと区別している。

このような機構を用いることでカーネル空間とユーザ空間での協調動作が可能になるが、事象の通知をアップコールによって行うため、不必要なモードの切り換えが生じるという問題があり、これを猪原らの研究 [49] では解決している。さらに従来の協調スレッドの実現方式を推し進め、岡坂らの研究のようには大域的なスケジューリングを可能にしているシステムもある [43]。商用 OS としては SunOS5.0[28] などがある。また，最近の情勢で言えば，FreeBSD の新しいバージョンではこの機構を取り入れているようである．[10]

スレッドモデルについての考察

表 2.2 にスレッドモデルによる利点と欠点をまとめた．2 レベルスレッドモデルはユーザレベルスレッドモデル，協調型スレッドモデルはユーザレベル，2 レベルスレッドモデルの拡張であるため，協調モデルがもっとも性能がよいと言える．

表 2.2: スレッドモデルの比較

	スケジューラの数	スレッド制御効率	スレッドの並列実行	OS からの通知	実装のしやすさ
Kernel Level	1	×	○	○	○
User Level	2	○	×	×	○
2 Level	2	○	○	×	△
協調型	2	○	○	○	×

マルチスレッドアーキテクチャにおけるスレッドモデル

一般的なスレッドモデルの考察は前述したとおりであるが，本研究の主眼であるマルチスレッドアーキテクチャプロセッサへの適用はどのようにすればよいだろうか．

前述したとおり，計算実体の管理が非常に軽量にできるのがマルチスレッドアーキテクチャの可能性の一つである．そのため，この効果を十分に発揮するためにはカーネル遷移などのオーバヘッドが必要になるカーネルスレッドモデルは不向きであるといえる．

しかし，従来のオペレーティングシステムではカーネルのみが物理的な計算実体を管理していた．そのため，ユーザレベルでそれらを行うものは存在していない．

プロセス・スレッドスケジューリング

複数のスレッド，もしくはプロセスがあったとき，それを計算実体にたいしてどのように割り付けるかが問題になる．SMP 構成システムでは，計算実体は各プロセッサであったため，どのプロセスを同時実行させるべきだ，などの研究はあまりされていなかったが，メモリ割り当て戦略やキャッシュヒット率などを効率よくするための研究はいくつか存在する．[45]

しかしマルチスレッドアーキテクチャではプロセッサの計算資源を共有することがあるため，同時実行する計算実体のスケジューリングは非常に重要になる．これに関する研究もいくつかあるが，[26] [27] [21] 共通するアプローチとして実際の実行結果（プロファイル結果）をもとにスケジューリング戦略を組みなおすという，フィードバックを利用したシステム構成となっている．

また，どのようにプロファイル結果を反映するかについても，並列度を下げる方式 [38] や並列実行する計算実体を選択する方法などがある．

2.2.2 排他制御・同期機構

マルチスレッドアーキテクチャは，演算器やメモリをを共有するするので，それら同士の競合が起こる可能性がある．特に一つの演算器に処理が集中し，競合が発生すると実行性能のボトルネックとなる可能性がある．そのため，スレッド制御でも，極力メモリアクセスなど，処理を集中させないことが求められる．

スレッド制御として，排他制御・同期機構を実装する場合には，従来のスレッドライブラリなどではスピンロックを行うか，スレッド切り替えを用いて実装するのが一般的である[*1]．

スピンロックとは他のスレッドがロックを解放するまで繰り返しロック変数を監視する手法である．スピンロックではロックが解放されるとすぐにロック獲得の処理へ移ることができるという利点がある．しかし，ロック変数が格納されているメモリ参照を繰り返すため，マルチスレッドアーキテクチャにおいてはメモリアクセスが頻発し，他の計算実体の実行を妨げる可能性がある．

スレッド切り替えは，他の待ち状態のスレッドへ処理を移すことで全体の性能を向上させることができる．しかし，この方式は実行コンテキストの復帰と退避を行うため，オーバーヘッドが大きい．そのため，できるならば他の方式による排他制御，同期機構を実現することが望ましい．

文献 [24] では，SMT アーキテクチャにおけるスピンロックについていろいろな負荷とプロセッサ構成による比較を行っている．

2.2.3 メモリ管理

メモリ管理はオペレーティングシステムの重要な課題のひとつである．とくに，仮想メモリをサポートしているシステムではその管理戦略がシステム全体の性能に大きく関わってくる．

マルチスレッドアーキテクチャにおけるメモリ管理としては [17] などで提案されている．

2.3 プログラミング言語処理系の課題

プログラムを記述したソースコードを実際に実行するためには言語処理系を通さなければならない．いくつか方法があるが，ここではコンパイラとインタプリタという視点でマルチス

[*1] もしくはその両者を組み合わせるアダプティブロックが考えられるが，これらと同様の問題点が発生するため本論文では言及しない．

レッドアーキテクチャプロセッサにおけるそれらの課題について述べる．

2.3.1 コンパイラ

コンパイラは一般にあるプログラムソースを入力すると他の形に変換して出力するプログラムである．多くはその出力はある計算機システム上で実行するための機械語列であったりするが，他のプログラム言語記述に変換されて出力する場合もある．

現在主流の RISC（Reduced Instruction Set Computer）プロセッサアーキテクチャはプロセッサの構成をシンプルにし，複雑な最適化をなるべくソフトウェア，具体的にはコンパイラに任せることで全体の性能を向上させようというアプローチであるため，今日でのコンパイラの責任は非常に大きい．

ここではマルチスレッドアーキテクチャ向けのコンパイラの課題として次の 3 つを述べる．すなわち，自動並列化，最適化，投機スレッドである．

自動並列化

並列処理システムにおいて，その性能を一つのアプリケーションが有効に活用するためにはそのアプリケーション自体を並列実行できるようプログラムを記述しなければならない．しかし，一般に並列プログラムを行うのは困難な場合が多い．

これは，プログラムやアルゴリズムが本質的に逐次実行するモデルである場合顕著である．このようなプログラムを並列化するためにはアルゴリズムなど自体を再考しなければならない場合が多い．そのため，コンパイラによる自動並列化が必要になる．

自動並列化コンパイラはプログラム中の並列実行可能な部分を解析し，その並列計算機システムで効率よく動作するプログラムを生成するコンパイラである．主なアプローチとして，まったくの逐次実行型プログラムを解析して並列化する手法とプログラミング言語を変更し，並列化のヒントを与えるようにしたり，自然に並列処理を記述できるようにしたものがある．

OpenMP[2] は C や Fortran のプログラム中に特殊なディレクティブを挿入し，並列化可能なことをコンパイラに伝えることを目的とした規格である．コンパイラはそのディレクティブによってヒントを得，効率よく並列化できる部分であれば，それをその計算機システムでの並列化制御機構用コードを作成する．

マルチスレッドアーキテクチャにおいて，このアプローチは現実的なものと考えられる．つまり，既存の処理系の並列単位の解析をプログラマにある程度押し付けることで，その解析コストを安価にできるからである．しかし，マルチスレッドアーキテクチャ専用のこの処理系は

確認できていない．

Cilk[5] や OPA[46]，Schematic[31]（と，これを効率的に実行させるための細粒度並列スレッド管理機構 [30]）などは細粒度並列処理を記述しやすいよう設計されたプログラミング言語である．これらは，関数呼び出し（または言語上それにあたる機構）をすべてスレッド生成として実現している．

このアプローチはマルチスレッドアーキテクチャにとって非常に興味深いが，これをサポートした処理系は存在しない．言語レベルで細粒度並列性を抽出でき，また，マルチスレッドアーキテクチャは従来の SMP 構成などの並列システムよりも細粒度並列性を生かすことができるアーキテクチャである．そのため，たとえば抽出できた細粒度並列単位をマルチスレッドアーキテクチャに適した手法で管理，制御できれば効率よく実現できるはずである．

完全な自動並列化のアプローチについては，古くはベクトル化コンパイラなどで研究されてきたループ並列化の研究や，タスク並列化の研究など，豊富に存在する．

また，並列単位の抽出以外にも，その並列単位をどのように管理するかも問題になる．たとえばマルチグレイン並列化コンパイラ [47] では，並列単位がそれぞれキャッシュメモリを備える構成を前提とし，全てを静的にスケジューリングすることでキャッシュヒット率を向上し，全体のスループットを向上することを可能にしている．しかし，このモデルでは静的にスケジューリングを行うため，割り込みなどのスケジューリング撹乱要因が発生したときには問題になる．

文献 [23][18] によると，既存のループ並列化の技術についても SMT アーキテクチャでは有利なもの，不向きなものがあるという調査報告を行っている．たとえば，ループ並列化を行ったとき，どのようにそれを配分するか，という問題があるが，従来の SMP 構成用プロセッサ向け並列化の場合，メモリアクセスのローカリティをあげるように並列化していた．しかし，SMT アーキテクチャではキャッシュを共有することが一般的なので，この方針では逆に性能低下の原因になることが示されている．マルチスレッドアーキテクチャむけ並列化システムは，これらの要因を考慮して並列化を行う必要がある．

また，文献 [23] では，同時実行する並列度の最適値について考察している．

最適化

RISC アーキテクチャ向けコンパイラ最適化は数多く提案されている．マルチスレッドアーキテクチャでも，ベースとなるアーキテクチャは通常の RISC アーキテクチャである場合が多いのでこれらの手法はそのまま利用できる場合が多いが，しかしその高速化のための最適化が性能低下を招く原因となる場合もある．

文献 [18] によると，SMT プロセッサ上ではたとえばソフトウェアパイプライニング（つまり，パイプラインを埋めるために命令列をスケジューリングしなおして配置すること）やソフトウェアスペキュレーション（を適用した最適化後のソースは SMT アーキテクチャでは効率的に実行されないことが示されている．これは，これらの最適化のオーバヘッドが悪影響を及ぼしているということである．それらの最適化のためのプロローグコードの実行を行うよりも，他の計算実体の実行を優先させるべきであることを示している．

投機実行

マルチスレッドアーキテクチャの計算実体のいくつかを投機実行のために利用しようというアプローチがある．プロセッサの投機実行と同様のことをソフトウェアで行おうというものがある．

2.3.2 インタプリタ

インタプリタは実行にコンパイルという段階を明確に行うことなくプログラムを実行するための処理系である．そのため，実行が手軽に行え，プログラムの記述と実行のターンアラウンドタイムが短くなり，プロトタイピングなどの用途に便利である．また，多くの場合，この方式ではリフレクションや高階関数のような手続き自体をファーストクラスオブジェクトとできたり，いわゆる `eval` のような，実行時に新たにプログラムを生成するような処理も行うことができる．このような抽象度の高い手法が利用できるため，プログラムが表現力の高いものになる．抽象度が高いプログラミング言語を実行するにはそれなりのオーバヘッドがかかるが，最近の計算機の高性能化にともない，これが現実的に利用できるようになってきた．そのため，今後ますます利用される処理系の方式であると思われる．[*2]

インタプリタでの処理を高速化するためには並列計算機上で並列処理として行うようにすることがもちろん有効である．

現状では，プログラミング言語がスレッド（もしくはそれに類する概念）をサポートしている場合，それを OS やライブラリが提供するスレッド機能に割り付けて並列実行する手法が利用されている．Java[1] や Python[3] などのプログラミング言語のある処理系ではこのような手法が主に利用されている．

[*2] 筆者自身としては，今後コンパイラを必要とするような低レベルな言語（計算機アーキテクチャよりの言語）は利用されなくなると思っている．つまり，すべてのプログラマはインタプリタ処理系を利用するのではないだろうか．唯一低級言語を利用するプログラマはシステムソフトウェア作成者だけであると思っている．

しかし，逐次プログラムを自動的に並列に実行させるようなアプローチは現状ではあまり研究されていない．これは，そもそも高速化を目的としたプログラムならば並列化コンパイラなどの既存の研究成果を利用すればいいのではないか，という考えが大勢を占めているからだと思われるが，今後インタプリタがより利用されてくるにつれ，このインタプリタ自身による逐次プログラムの並列実行は重要になると思われる．

文献 [47] では，投機並列実行をインタプリタで行うことで逐次プログラムを並列実行するインタプリタの構想を示している．また，文献 [37] や文献 [35] では Java 仮想マシン [36] におけるループ並列化手法を提案している．

しかし，マルチスレッドアーキテクチャの軽量な計算実体の管理という特徴を利用すれば，これらがもっと効率的に実現できると思われる．

2.4 本章のまとめ

本章では，マルチスレッドアーキテクチャプロセッサの特徴を示し，それによりシステムソフトにはどのような課題があるかを示した．

マルチスレッドアーキテクチャの従来の並列システムとの相違，とくに SMP システムとの相違は，計算資源をより共有するということであり，たとえばキャッシュメモリや演算器，演算用パイプラインなどを共有することである．また，それらを共有することで競合が発生し，新たなボトルネックになる可能性がある．また，オンチップに複数の計算実体が動作するため，従来はある程度のオーバヘッドが必要であった通信コストが非常に小さくなる．たとえば，キャッシュメモリを共有している場合，その共有しているキャッシュをとおして他の計算実体と情報をやり取りすることができる．また，計算実体の管理，制御も今まで以上に軽量に行うことができる．

システムソフトウェアは，これらの性質を踏まえた上で最適なものを提供しなければならない．

オペレーティングシステムやスレッドライブラリは，そのモデルの考察が必要である．とくに，計算実体の管理はプロセッサによる軽量な管理を生かすにはユーザレベルでこれを行うようにしたほうが効率がよい．また，計算実体のスケジューリングが重要になる．

排他制御や同期などは，従来のスピンロックではまずい．

言語処理系はキャッシュメモリの共有などや，安価な計算実体の管理を建設的に利用するものでなければならない．

第 3 章

目標とするシステムとその全体構成

前章で述べた課題を踏まえ，我々の研究グループでは次のような計算機システムを目標とする．

- オンチップマルチ SMT プロセッサ
- オペレーティングシステムとユーザレベルスレッドライブラリ
- プロセッサに適用可能な言語処理系（コンパイラ・インタプリタ）

これを全体像として図示したのが図 3.1 である．図には各パーツに固有名を示しているが，それぞれについて以下に紹介する．

3.1 プロセッサ

集積度を上げてさらに並列度を上げるため，構成を SMT プロセッサをさらに 1 チップ上に複数集積する．つまり，CMP の各プロセッサエレメント（PE: Processor Element）がそれぞれ SMT プロセッサとして動作するプロセッサを指向する．このプロセッサを *OChiMuS Processor* と呼ぶ．

以下，SMT プロセッサ上の並列実行する計算実体をそれぞれ実スレッド（AT, AThread: Architecture Thread）と呼ぶが，この各 PE の実スレッドの管理は実行権限がユーザレベルからでも行えるようにする．これは，より効率的な実スレッド管理を可能にするためである．

また，OChiMuS プロセッサの各 PE も独立したプロセッサとして実現できるようにしており，それぞれ OChiMuS PE プロセッサとして設計されている．これについては次章で詳しく述べる．

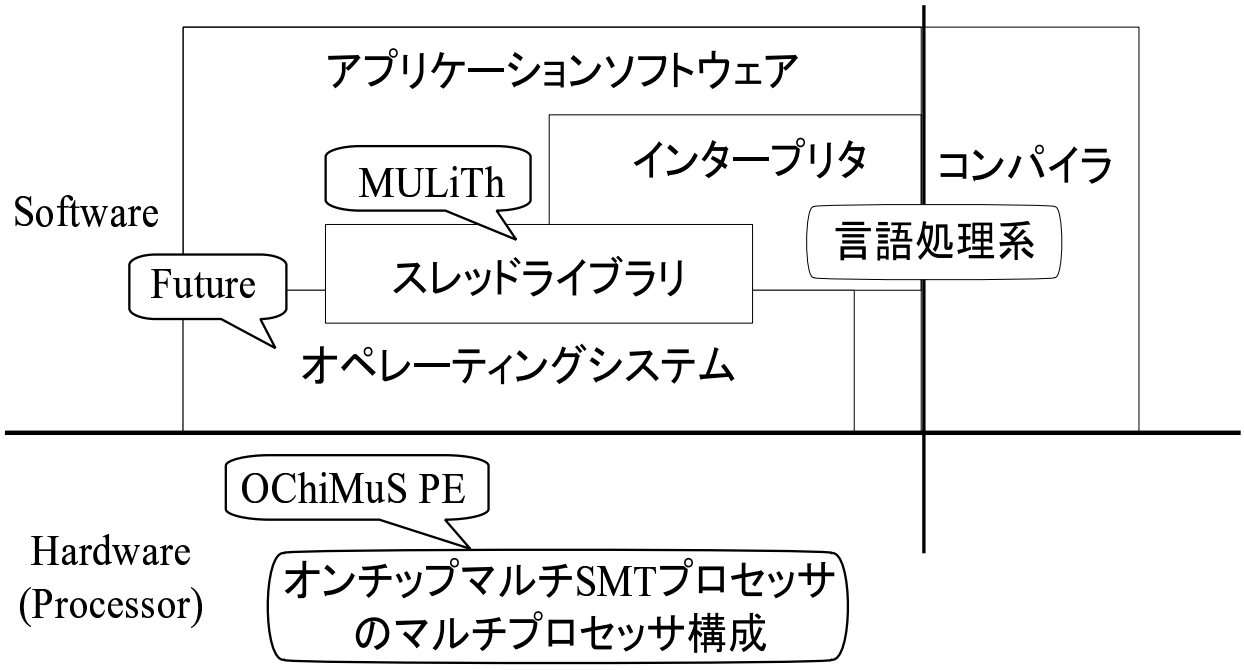

図 3.1: 目標とするシステムの全体像

3.2 オペレーティングシステムとスレッドライブラリ

プロセッサが提供するユーザレベルによる軽量な実スレッド制御機構を生かすため，オペレーティングシステムは実スレッド管理を行わないで，ユーザレベルがそれを行うようにする．これを実現するためのプロセス管理を行うオペレーティングシステムを *Future* という．

Future は一つのプロセスにひとつのプロセッサを割り当てる．これにより，複数プロセス同時実行することによるワーキングセット増大を防ぐこれができる．

ユーザレベルスレッドライブラリを提供し，そのライブラリがいわゆるソフトウェアのスレッドを管理し，実スレッドを直接制御することでそのスケジューリングを行う．このスレッドライブラリを *MULiTh*（User level Thread Library for Multithreaded architecture）という．

実スレッド管理をユーザレベルで行うことで，軽量な実スレッド制御を可能にする．また，Future と MULiTh が連携することで効率的な OS からの事象通知を可能にする．

3.3 言語処理系

OChiMuS プロセッサの構成に適した言語処理系を作成する．すなわちユーザレベルで可能な実スレッド管理を利用した自動並列化コンパイラや，共有するキャッシュメモリなどを意識したメモリ配置を行うような最適化アプローチなどを行う．

3.3.1 コンパイラ

自動並列化コンパイラはキャッシュメモリを共有することで建設的な干渉を起こようにアクセス対象となるメモリを分散する．また，軽量な実スレッド制御を利用した計算単位の生成を新たに考案する．

最適化についても同様にマルチスレッドアーキテクチャに適したものを用意する．

3.3.2 インタプリタ

マルチスレッドアーキテクチャにおけるインタプリタは，上述したオペレーティングシステムおよびスレッドライブラリが提供するスレッド機能を利用したものが容易に準備できる．また，これも軽量な実スレッド制御を利用したより細粒度の並列化，たとえばインタプリタの仮想命令列数個ずつに関する並列化を行い，逐次プログラムの並列実行を実現する．

第 4 章

システム設計詳細

SMT プロセッサである OChiMuS PE，オペレーティングシステム Future，スレッドライブラリ MULiTh について設計が終わり，OS とライブラリについては実装も行った．本章ではそれらについて詳述する．

なお，筆者が作成した MULiTh と，MULiTh と Future の連携機構についてはより詳細にその設計を示す．

4.1 システムの現状と全体構成

この目標を満たすため，現在我々の研究グループでは図 4.1 のようなシステムを設計するに至っている．

現状ではプロセッサは CMP 構成ではなくその PE である OChiMuS PE までの設計が終了している．

OChiMuS PE を効率的に利用するためのオペレーティングシステム Future とそれと対になるユーザレベルスレッドライブラリ MULiTh も設計，実装を行われている．

言語処理系は今後の課題である．

4.2 OChiMuS PE プロセッサ

本節では，東京農工大学工学部情報コミュニケーション工学科中條研究室で開発中のオンチップマルチスレッドプロセッサアーキテクチャである OChiMuS PE プロセッサ [44] について説明する。このプロセッサは、基本的に MIPS プロセッサアーキテクチャをベースにし、それにマルチスレッドをサポートするためにのモジュールや命令を新たに拡張している。

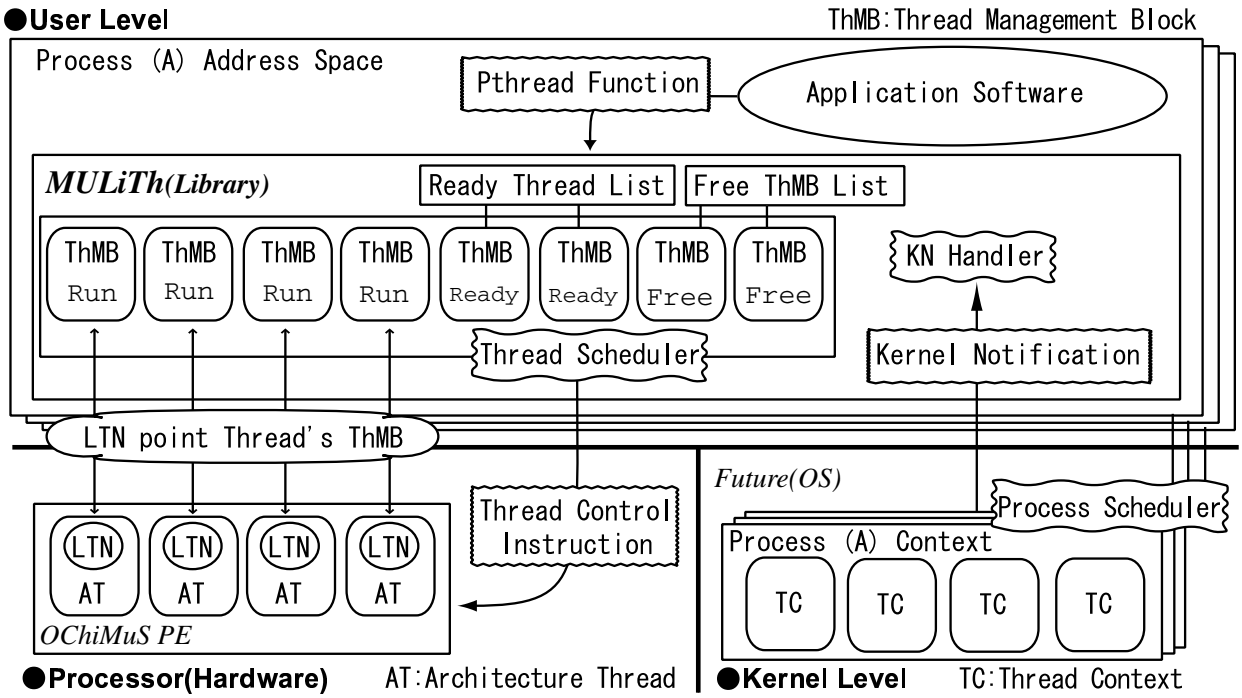

図 4.1: 現状のシステムの全体像

OChiMuS PE は，将来的にはこのプロセッサを 1 エレメント (PE:Processing Element) としてこの PE を同一チップ上に複数搭載したオンチップマルチ SMT(OChiMuS) アーキテクチャとすることを検討しており，現在本学中條研究室で研究開発が進められている．このため，ここで示すアーキテクチャによるプロセッサを OChiMuS PE と称す．

4.2.1 プロセッサ概要

OChiMuS PE はマルチスレッド化を実現するために複数の実スレッドを持つ．また，実スレッドは次に示すリソースを個別に持つ．

- プログラムカウンタ
- 実行状態レジスタ (PCSR)
- 論理スレッド番号レジスタ (LTNR)
- 汎用レジスタファイル

図 4.2 では，一つの実スレッドがどのような構成となっているかを示す．

論理スレッド番号レジスタはその実スレッドの**論理スレッド番号**(以下、LTN) を格納するレジスタである。この論理スレッド番号で示されたスレッドを” 論理スレッド” と称する。論

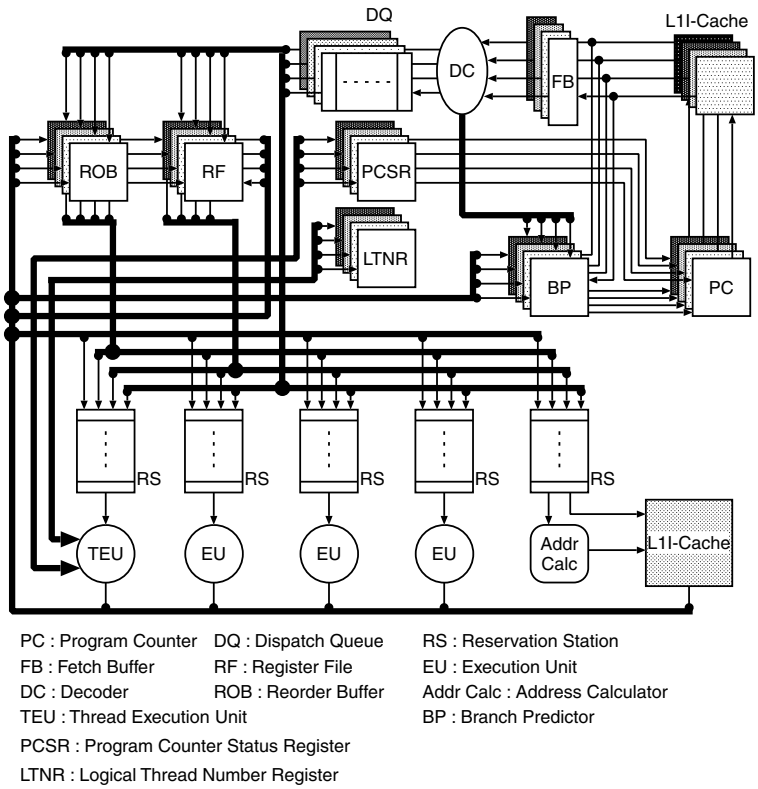

図 4.2: OChiMuS PE の構成 (元中條研究室/現 NTT シリコンシステムズ 河原氏提供)

理スレッドは、プロセッサの実スレッドを仮想化したものと考えられ、これにより、システムソフトウェアと連携した柔軟なスレッド操作が可能になる。論理スレッドをプロセッサ内に割り当てるときに、同時に LTN を指定し、これがこのレジスタに格納される。プロセッサはこの LTN を参照して、スレッド制御命令の実行を行う。

実行状態レジスタ (PCSR) は、その実スレッドが現在どのような実行状態にあるかを示すレジスタであり、次のような状態がある。

- PCS_NORMAL：実スレッドは命令フェッチ可能
- PCS_BLOCK：実スレッドは命令フェッチを一時停止
- PCS_HALT：実スレッドは完全停止

実行状態の状態遷移は図 4.3 によって示され、図中の矢印の隣に記述しているスレッド制御命令によって状態を遷移する。

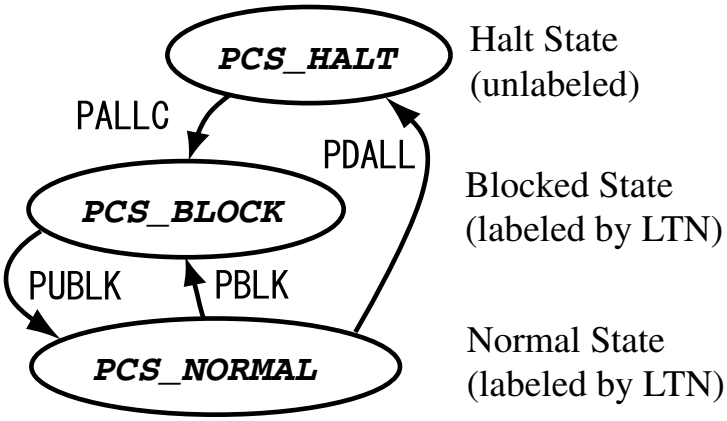

図 4.3: OChiMuS PE プロセッサの実スレッドの状態遷移

4.2.2 プロセッサ内部のスレッド管理

OChiMuS PE プロセッサでは、LTN を用いることでプロセッサアーキテクチャレベルでのスレッド制御の仮想化を可能にしている。プロセッサのスレッド制御を LTN を用いた論理スレッドへの操作であるとすることで、実スレッドを直接指定するスレッド制御よりも容易なスレッド管理が可能になる。例えばプロセッサ内でいくつの実スレッドが実装されているか、現在実際に実行している LTN はなんであるか、の管理を行うことなしにスレッドの管理を行える。

LTN を用いない場合、スレッド制御を実行するたびに毎回メモリにあるシステムソフトウェアのスレッド管理表などでどのスレッドが割り当てられているか、どの実スレッドにどのスレッドが論理スレッドとして実行中であるか、などの情報を調べる必要がる。オンチップマルチスレッドアーキテクチャでは、プロセッサ内で並列実行する実スレッドを増やすと、それに比例してメモリアクセスが頻発し、メモリアクセス自体が実行性能のボトルネックになる可能性があるので、それを抑える LTN の利用はオンチップマルチスレッドアーキテクチャの性能向上に有効であると考えられる。

そして、システムソフトウェアはスレッドの管理を LTN をキーとして行うことにより、システムソフトウェアでのスレッド操作をプロセッサのスレッド制御命令を利用して実行することが非常に容易になる。

表 4.1: スレッド制御命令

命令書式	説明
PALLC dr,sr0,sr1	実スレッドに論理スレッドを割り当てる
PDALL dr,sr	実スレッドに割り当てられている論理スレッドを開放する
PBLK dr,sr	実スレッドに割り当てられている論理スレッドを一時停止させる
PUBLK dr,sr	実スレッドに割り当てられている論理スレッドの一時停止を解除する
FWD dr,sr0,sr1	実スレッドに割り当てられている論理スレッドにレジスタの値を設定する

4.2.3 スレッド制御命令

スレッド制御命令は表 4.1 のとおりである。表 4.1 で示すように、基本的にソースレジスタで LTN を指定し、その実行結果（成功・失敗）がデスティネーションレジスタに格納される。つまり、スレッドの制御をプロセッサが失敗した場合にはソフトウェアがそのサポートを行えるようになっている。

PALLC 命令は論理スレッドをプロセッサ内に割り当てる時に用いる。この命令は、2 つのソースレジスタを用い、それぞれ生成する論理スレッドの LTN の指定とその論理スレッドの開始位置アドレスを指定する。もしプロセッサ内に空き実スレッドが存在する場合は、論理スレッドはプロセッサ内に割り当てられ、成功の意味の値がデスティネーションレジスタに格納される。もしなければ、失敗を意味する値がデスティネーションレジスタに格納される。

PDALL 命令は、指定した LTN を持つ論理スレッドが、もしプロセッサ内に割り当てられていればその割り当てを解除する。つまり、実行状態レジスタが PCS_HALT となり、それが意味する状態へ遷移する。指定した LTN を持つ論理スレッドが存在しない場合はデスティネーションレジスタに失敗を意味する値が格納される。

PBLK・PUBLK 命令は、指定したスレッドの一時停止、解除を行う。つまり、指定した LTN を持つ論理スレッドの状態を、PBLK 命令では PCS_BLOCK に、PUBLK 命令は PCS_NORMAL の状態に遷移させる。LTN で指定する対象の論理スレッドがない場合、または移行できる状態ではない場合[*1]、デスティネーションレジスタによってソフトウェアは成功、失敗を知ることができる。

[*1] PCS_BLOCK 状態のスレッドに PBLK 命令を実行するなど。

FWD 命令は、LTN で指定した論理スレッドの指定したレジスタに、値をセットする命令である。セットできる対象の実スレッドは PCS_BLOCK でなければならない。指定した論理スレッドがない場合、または PCS_BLOCK でないときはデスティネーションレジスタに失敗した意味の値を格納する。この命令により、プロセッサレベルでのスレッド間通信を実現することができる。

これらのスレッド制御命令は、前述した方針のとおり、ユーザレベルで実行可能な命令となっている。また論理スレッド番号レジスタ、実行状態レジスタもユーザレベルで利用可能となっている。

4.3 オペレーティングシステム Future

前節で説明したプロセッサを管理するために必要な OS アーキテクチャについて考察する。

Future は本研究室で開発中の OChiMuS PE 用のオペレーティングシステムである。マルチスレッドアーキテクチャに適したプロセスモデルをサポートし、本論文で述べるスレッドライブラリと協調動作することで高性能を目指す。

Future では UNIX の従来の API を可能な限り維持しつつ、カーネルの内部構造を OChiMuS PE プロセッサ向けに見直す方針とする。そして、できる限りカーネルのオーバーヘッドの軽量化を図ることを目標としている。すなわち、CPU 資源管理、メモリ管理、I/O 管理において、個々の制御機能をできる限りシンプルに設計し、カーネルで行うべき処理、ランタイムライブラリ、スレッドライブラリで行うべき処理を切り分け、そしてカーネル内処理のオーバーヘッドを最小限にとどめらるよう小型・軽量な OS を目指している。

4.3.1 プロセス管理

Future では，プロセスは各種資源を割り当てる単位であり，プロセッサを仮想化したものと定義される．資源はたとえばアドレス空間であり，I/O 資源である．Future で仮想化するプロセッサとは，OChiMuS PE 全体であり，各実スレッドの仮想化はスレッドライブラリが行う．このプロセスが複数存在し，カーネルがこれらのプロセスを切り替えて実行する．

Future によるプロセス切り替えは次の手順で行う．すなわち，動作中のプロセスで実行中の実スレッドのコンテキストを退避する。そして切り替える対象のプロセスが、前回プロセス切り替えのときに退避していた実スレッドのコンテキストの復帰，である．複数存在する実スレッドはアドレス空間を共有するので，切り替えは同時に実行される．

4.3.2 スレッドライブラリとの連携

スレッドの管理はユーザレベルのスレッドライブラリが行うが、ユーザレベルで解決することが困難な問題は OS からスレッドライブラリへ通知を行うことで、これを解決する．本研究ではこのような協調型スレッドライブラリを目指す。これを実現する機構を Kernel Notification という。

Kernel Notification ではカーネル内でスレッドがブロックする必要があるときなどの事象をスレッドライブラリにアップコールにより知らせる機構であり，従来の Scheduler Activations[4] よりも効率的にこれを実現する．Kernel Notification の詳細は後述する．

4.4 スレッドライブラリ MULiTh

MULiTh はスレッド処理機構を提供するユーザレベルスレッドライブラリである．

図 4.1 は，提案するシステムの全体構成を示している．従来のスレッドライブラリとは違い，プロセッサ，つまり複数の実スレッドをスレッドライブラリが管理する．これにより，効率的なスレッド管理を可能にする．

スレッドは実スレッド数以上生成可能であり，スレッドライブラリはスケジューリングによりスレッドを複数の実スレッドに割り当て，それらを並列実行させる．

プログラミングインターフェースは，POSIX スレッド (Pthread)[14] の仕様に準拠したものを提供する．Pthread はスレッドライブラリの規格としては標準的なものであり，アプリケーションソフトウェアのソースレベルでの互換性を提供する．

Pthread 関数により，スレッドの生成，削除，排他制御，同期機構などを提供する．スレッドの制御は，OChiMuS PE のスレッド制御命令を利用することで，高速に行う．

また，OS と協調して動作することで，カーネル内のブロッキングの問題について回避し，プリエンプションのあるスレッド管理を実現している．これについては次節で述べる．

4.4.1 プログラミングインターフェース

スレッドライブラリ MULiTh のプログラミングインターフェースは、POSIX Thread[14][16](以下、Pthread） の仕様に準拠するものとして設計を行った。これは、C 言語で記述できるスレッドライブラリの仕様としては、標準的な存在であり、MULiTh 用に作られたプログラムでもその他の Pthread をサポートする処理系で動作させることができる。また、逆に Pthread を利用してマルチスレッド化したソフトウェアは、MULiTh を利用して

表 4.2: MULiTh で利用できる主な Pthread 関数

Pthread 関数名	説明
pthread_create	スレッド生成
pthread_exit	スレッド終了
pthred_join	スレッドを合流する
pthread_mutex_lock	排他制御（ロック獲得）
pthread_mutex_unlock	排他制御（ロック解除）
pthread_cond_wait	同期（合図を待つ）
pthread_cond_signal	同期（合図を送る）
pthread_self	現在実行中のスレッドのスレッド識別子を得る

動作させることができる。スレッドライブラリの仕様としては、Pthread が一番低レベルなスレッド操作が可能であり、本研究の目的であるマルチスレッドアーキテクチャのための基本的，汎用的なスレッドライブラリの仕様として適している。

実装した主な Pthread 関数は表 4.2 で示し，図 4.4 で示す宣言に基づいて利用する．

これらはたとえば次のように利用する．このプログラムでは，pthread_create によって

```
/// スレッド生成
int pthread_create(pthread_t *th,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),void *arg);

/// スレッドを終了する
void pthread_exit(void *retval);

/// スレッドを合流する
int pthread_join (pthread_t th, void **exit_status);

/// 条件変数を待つ
int pthread_cond_wait(pthread_cond_t *,th_mutex_t *);

/// 排他制御用　ロック
int pthread_mutex_lock(pthread_mutex_t *mutex);

/// 排他制御用　ロック解除
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/// 条件変数に合図を送る
int pthread_cond_signal(pthread_cond_t *);

/// スレッド ID 取得
th_t pthread_self(void);
```

図 4.4: 作成した Pthread 関数のプロトタイプ宣言

```
void func(int n){
  static int val;
  if(n&1){
    pthread_exit(&n);
  }
}

void *thread_func(void *p){
  func(*(int *)p);
  return 0;
}

#define THREAD_NUM 10

int main(){
  pthread_t th[THREAD_NUM];
  int n[THREAD_NUM];
  int i;
  for(i=0;i<THREAD_NUM;i++){
    n[i] = i;
    pthread_create(&th[i],0,thread_func,&n[i]);
  }
  for(i=0;i<THREAD_NUM;i++){
    void *res;
    pthread_join(th[i],&res);
  }
}
```

図 4.5: スレッド生成，削除，合流を行うサンプルプログラム

新たにスレッドを生成する．また，pthread_exit によってその実行を終了する．最後に pthread_join によってその実行をスレッド生成側（Creator）と生成された側（Createe）で合流する．

4.4.2 スレッド管理

MULiTh では生成されたスレッドの状態の管理と生成するために必要なメモリ領域の管理を行う．図 4.6 では MULiTh が管理するデータについて示している．生成されたスレッドは，実行中のもの，待ち状態のもの，ブロック状態のもの，という状態を持っており，MULiTh はそのスレッドの状態に応じた処理を行う．たとえば，スケジューリングを行う際には待ち状態のスレッドのみを対象とする．また，スレッドの生成のために必要な空きのスレッド管理領域も管理する．すべてのスレッドが終了したとき，そのプログラムが終了したということになる．

各スレッドは，それぞれ**スレッド管理ブロック**(ThMB: Thread Management Block) を持つ．ThMB は，スレッドを中断したときのコンテキストの退避領域と，そのスレッドの属性を保存するための領域をもつ．現状では，ThMB はスレッドライブラリが静的に保持する．

MULiTh では，各スレッドの ThMB の先頭アドレスを，そのスレッド識別子として利用す

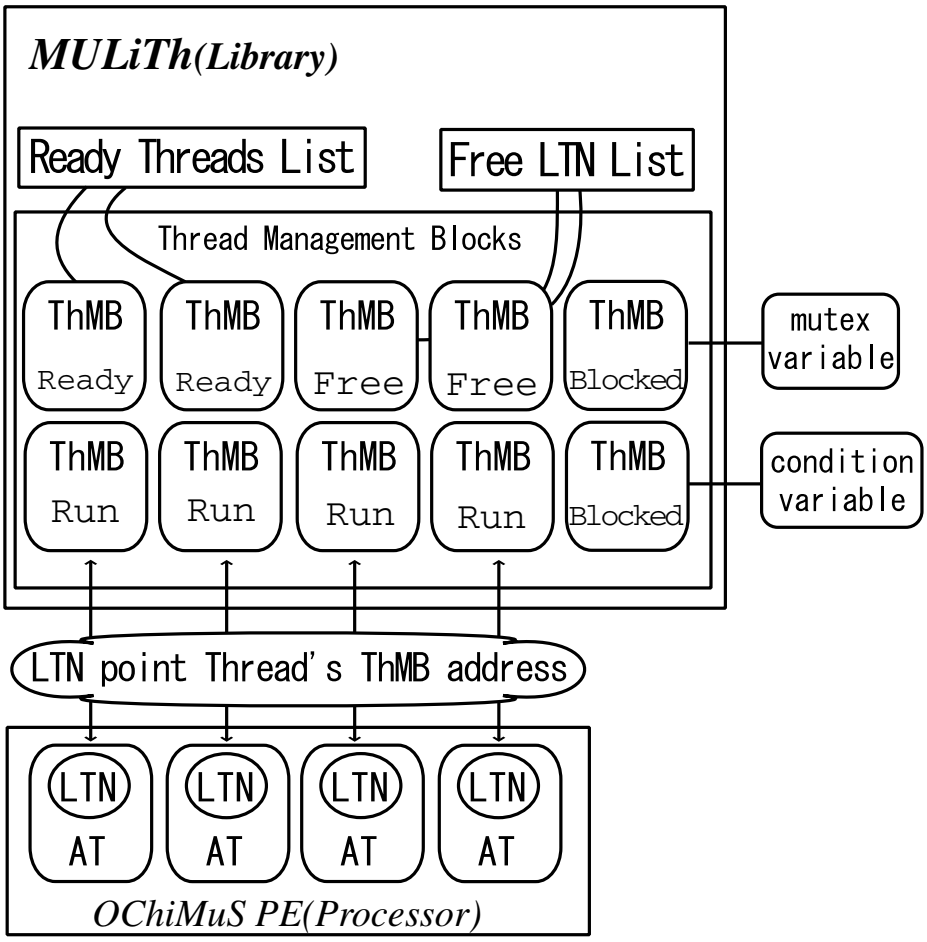

図 4.6: スレッド管理概観

る．各スレッドがそれぞれ個別の ThMB を保持するので，その先頭アドレスも互いに違うものとなるため，識別子として利用できる．

スレッド管理ブロックは、スレッドを中断したときの実行コンテキストを退避するための領域と、そのスレッドの属性を保存するための領域を持つ。この領域は、C 言語風に書くと次のようになる。

```
typedef struct{
  ThreadContext context;
  ThreadAttribute attr;
}ThreadManagementBlock;
```

スレッド識別子をプロセッサのスレッド制御命令に必要な LTN とする．こうすることで，ハードウェアの実スレッド管理とソフトウェアのスレッド管理を一元化することができる．ま

```
pthread_t pthread_self(){
  int ret;
  asm("mfc2 %0,$2"
      : "=r" (ret)
      );
  return ret;
}
```

図 4.7: pthread_self 関数の定義

た，LTN がスレッド識別子であると定義できる．LTN は，プロセッサの専用レジスタに格納されるため，レジスタアクセスと同等のコストで取得することができる．これより，自スレッドのスレッド識別子を取得する pthread_self 関数を軽量に実装することができる．これは，スレッド制御処理において必要なので，この取得が軽量に実装されるとスレッド制御処理全体が高速に実行できる．

図 4.7 では，その定義を示す．アセンブラの記述は GCC の拡張書式を利用している．自スレッドの ThMB へのアクセスは，スレッド制御処理において必ず行われるため，スレッド識別子の取得が軽量に実装されるとスレッド制御処理全体の性能が向上する．[*2]。

プロセッサの実スレッドが，LTN によりどのスレッドを実行しているか，という情報を管理しているため，MULiTh ではどのスレッドが実行中であるか，という情報は管理しない．プロセッサのスレッド制御命令を利用することにより，ソフトウェアで管理する場合と同様の効果が得られる．つまり，実行中でない場合，スレッド制御命令はエラーを返すため，そのときに実行中でないと判断することができる．

スレッドを生成したが、プロセッサの実スレッドに PCS_HALT 状態のものが無かった場合など、CPU 利用権を待つスレッド（**待ちスレッド**）が生じる可能性がある。そのようなスレッドを管理するため、待ちスレッドリストを用意する。この待ちスレッドリストは、なんらかのタイミングでスレッドのスケジューリングが発生したとき、たとえばあるスレッドの処理が終了し、そのスレッドが削除されるときなどに、待ちスレッドをその終了したスレッドが利用していた実スレッドに対して復帰させるために利用する。

待ちスレッドリストは，プログラム開始時は空であり，生成したスレッド数が実スレッドの数以上になった場合に格納される．

現状ではスレッドのスケジューリングは FIFO によって行っている。待ちスレッドリスト

[*2] LTN が 32bit の場合、このようにアドレスを代入できるが、現状ではハードウェアの要因から LTN が 16bit となる可能性も残っている。この場合は管理するスレッドコンテキストデータを配列として管理し、その配列のインデックスとして LTN を利用することで同様な管理が可能になる。

のデータ構造をキューによって実装しているため、待ちスレッドリストから取り出すことがすなわちスケジューリングを行ったということができる。

Pthread の仕様では，プライオリティのあるスレッドスケジューラが求められているが，実装コストおよびスケジューリングコストの兼ね合いで現在では実装していない．

また，SMT アーキテクチャは各実スレッドが演算器を共有して並列に計算を行うため，同時に実行するスレッドがどのような種類の計算を行うかによって性能に大きな影響を与える．文献 [26] などでは，同時実行するスレッドのいろいろ組み合わせを試し，同時実行する最適な組み合わせを図る手法を検討している[*3]．このような SMT アーキテクチャ向けの工夫を持ったスレッドスケジューラは我々の研究グループで開発を進めている．

4.4.3 スレッドの生成と削除

スレッド生成 スレッドを生成する処理，すなわち `pthread_create` 関数を実行すると，プロセッサの実スレッド割り当て命令 `PALLC` を実行する．これが成功すれば，生成されたスレッドは実スレッドとして実行される．失敗した場合，そのスレッドを待ちスレッドリストへ格納する．

手順としては，次のようになる．

1. 未使用の ThMB のアドレス (LTN) を一つ得る (失敗ならスレッド数制限による生成失敗)
2. `PALLC` 命令を実行する
3. (2) が成功ならば，`FWD` 命令で初期値の設定を行い，`PUBLK` 命令でそのスレッドを実行させる（生成成功）
4. (2) が失敗ならば，開始コンテキストを設定し，待ちスレッドリストへ格納する（生成成功）

図 4.8 は，スレッド `T1` がスレッド `T2` を生成する様子を示している．

図 4.8 の (i) は生成手順の (3) の動作を示す．`PALLC` 命令は，状態が *PCS_HALT* の実スレッドがあれば，それに対し指定した LTN に割り当て，その実スレッドの状態を *PCS_BLOCK* にする命令である．ここでは図中 `AT2` で示している実スレッドが *PCS_HALT* であるため，`PALLC` 命令は `AT2` を対象に割り当てを行う．`AT2` は，LTN を `T2` に設定され，*PCS_BLOCK* になる．スレッド `T1` はスレッド `T2` に対し，スレッド `T2` 実行開始時の初期設定として `FWD` 命令によりレジスタ値の転送を行う．その後 `PUBLK` 命令を発行し，スレッド `T2` の実行を開始する．こ

[*3] OS でのスケジューリングを対象としており，プロセッサのパフォーマンスカウンタの利用を前提としている．

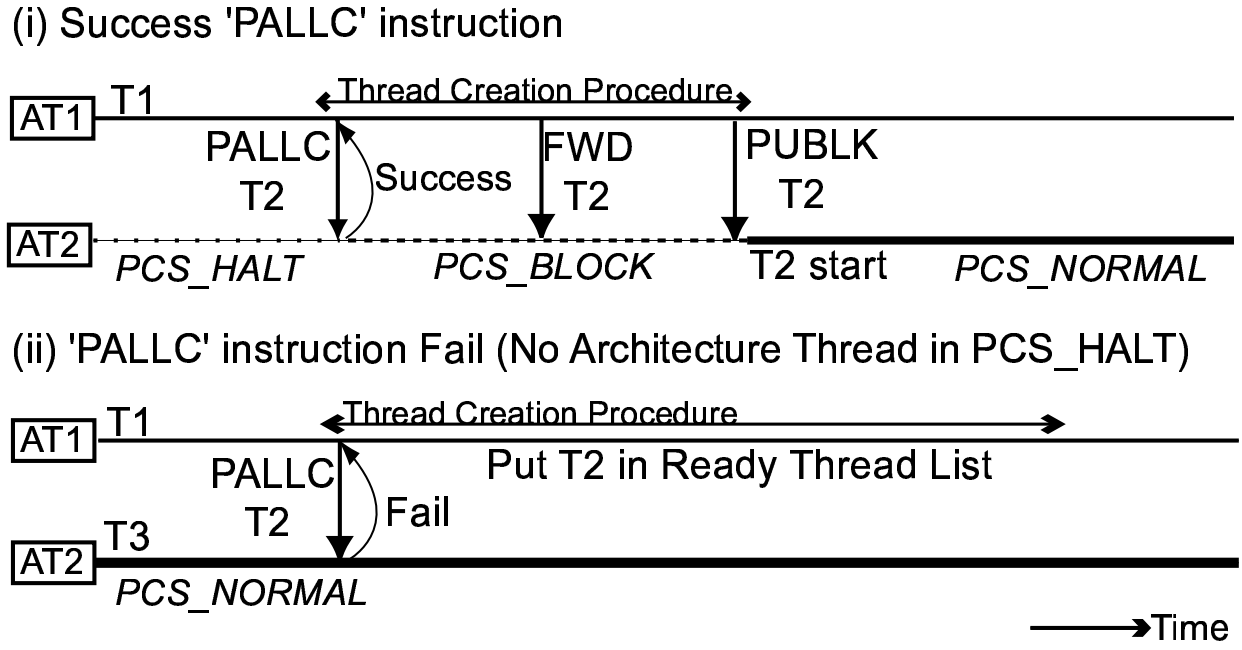

図 4.8: プロセッサ命令を利用したスレッドの生成

れにより，並列実行する実スレッドが一つ作られることになる．これらの制御はプロセッサ命令を利用して行うため，高速に実行することができる．

図 4.8 の (ii) は，(4) の動作を示しており，*PCS_HALT* 状態の実スレッドが存在しないため，PALLC 命令が失敗している．この場合，スレッド T2 の ThMB に実行開始のためのコンテキストを設定し，待ちスレッドリストへ格納する．これは，従来のユーザレベルで行うスレッド管理と同様である．PALLC 成功時に比べ，コンテキストの保存など，メモリアクセスが増える．

スレッド削除　スレッドの削除は，pthread_exit 関数が呼ばれるか，スレッドの開始ルーチンからリターンしたときに起こる[*4]．

削除処理は，次の順序で行う．

1. 空き LTN リストに自スレッドの LTN を格納
2. スレッドスケジューリング
3. スレッド切り替え

空き LTN リストに自スレッドの LTN を格納することで，その LTN および ThMB を再利

[*4] 厳密には，デタッチされたスレッドでない場合，そのスレッドが pthread_join によって合流された時点で削除が行われる．

用することができるようになる．次に動作させるスレッドをスケジューリングにより決定し，スレッド切り替えを行う．

次に動作するスレッドが存在しなかった場合（待ちスレッドがない場合），その実スレッドを PDALL 命令により停止する．ユーザレベルでこの命令を実行し実スレッドを停止するため高速に動作する．

自発的にスレッドを切り替えたい場合，つまり pthread_yield 関数ではスレッドスケジューラを起動することにより，他の待ちスレッドへ切り替わる．

また，スレッド生成，削除に限らず，スレッド制御は各実スレッドで並列に行うことを可能にしている．このために一部のクリティカルセクションでは MIPS プロセッサの LL，SC 命令を利用して排他制御を適宜行っている．

4.4.4 効率的な大量の細粒度スレッド生成処理

スレッドプログラミングのうち，多くの細粒度スレッドを生成する場合がある．これは，プログラムが本質的に並列性をもっている場合に，並列計算機上で処理速度を向上させるために行うことが多い．細粒度スレッドを生成するようなプログラムの場合，スレッドを大量に生成させる場合が多く，そのようなときにはスレッド生成のコストは出来る限り抑えたい．そこで，MULiTh では細粒度スレッド生成に適したスレッド生成方式処理をサポートする．

基本となるアイディアは簡単である．スレッド生成時，現状以上に並列性を上げられない場合，スレッド生成を失敗させ，スレッド生成者に逐次処理をさせるのである．我々は PALLC1 命令のみで実スレッドを実スレッドが作成できるかどうかを知ることができる．

細粒度スレッド生成時，通常のスレッド生成と同様に実スレッド割り当て命令 PALLC を実行する．このとき，スレッド割り当てが行うことができればスレッドは生成される．この命令が失敗した場合，通常のスレッド生成ではそのスレッドを待ちスレッドとしてスケジューリング対象とする．しかし，この待ちスレッドとして登録する作業にはオーバーヘッドがかかる．細粒度スレッド生成の場合は，このときの処理をスレッド生成数の限界による失敗 (EAGAIN) として細粒度スレッド生成者に返す．このとき，生成者は代わりの逐次処理を行う（図 4.9）．

また，スレッド生成に必要な未使用の ThMB のアドレス (LTN) の取得は，他スレッドとの同期処理が必要になるので，これもコストの大きい処理である．そのため，スレッド生成失敗のときは利用しなかった LTN をキャッシュしておき，次のスレッド生成のときに利用することにする（図 4.10）．

この工夫により，細粒度スレッドの実行時間を T_1，スレッド生成間隔を T_i とすると，T_1 が十分小さく，$T_i < T_1$ であるようなプログラム（一般的な細粒度スレッドを生成するようなプ

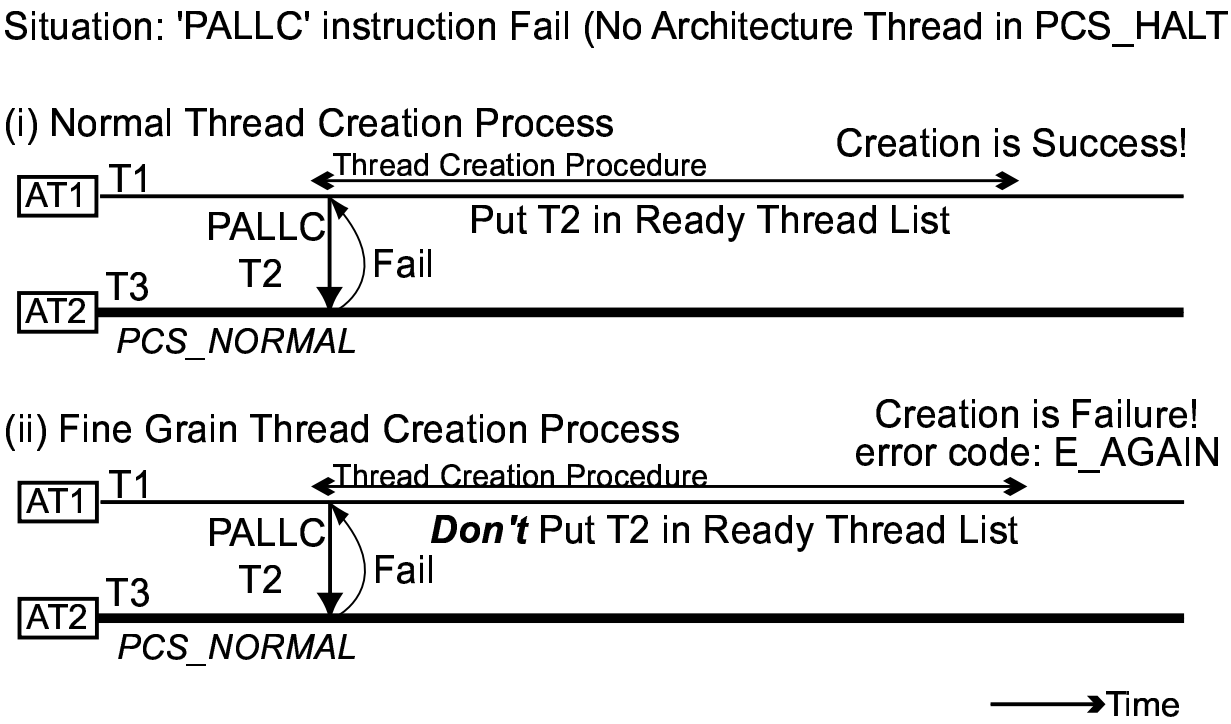

図 4.9: 細粒度スレッド生成処理

ログラム）では効率的にスレッド生成を行うことができる．この手法はマルチグレインスレッド生成，つまりいくつかの粗粒度スレッドと多くの細粒度スレッドが生成されるような場合でも有効である

しかし，本方式では，生成されたスレッド間が複雑な通信を行うようなプログラムには，スレッド生成が失敗した場合の処理を記述するのが難しくなるので不適切である．このような場合は通常のスレッド生成を行うようにする．ただし，細粒度スレッドを生成を行うようなプログラムの場合，この問題が起こるのはまれである．

4.4.5 排他制御，同期機構

スレッドの排他制御，同期機構ではスレッドの実行をブロックする場合がある．

1. `pthread_mutex_lock` 関数でロックが獲得できないとき
2. `pthread_join` 関数で対象スレッドが終了することを待つとき
3. `pthread_cond_wait` 関数で条件変数に対する合図を待つとき

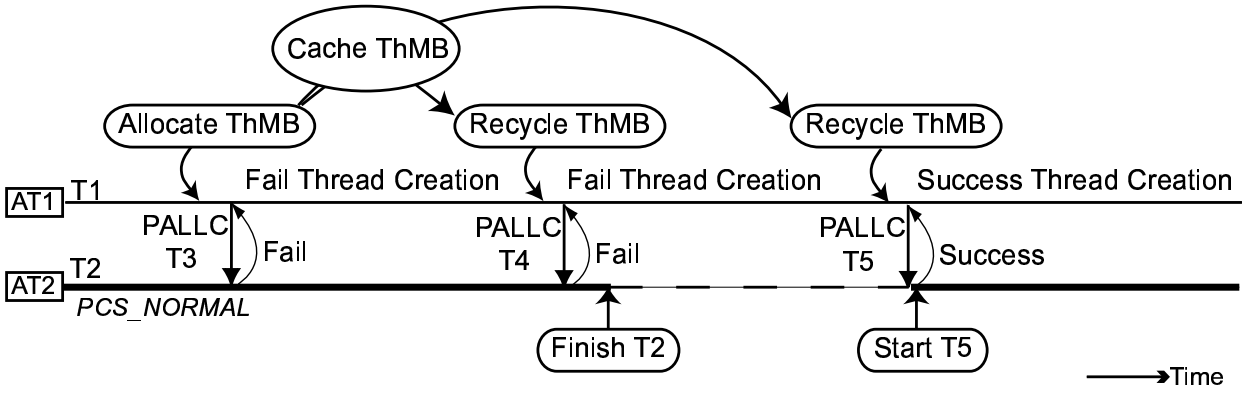

図 4.10: 細粒度スレッド生成処理の ThMB キャッシュを含めた動作

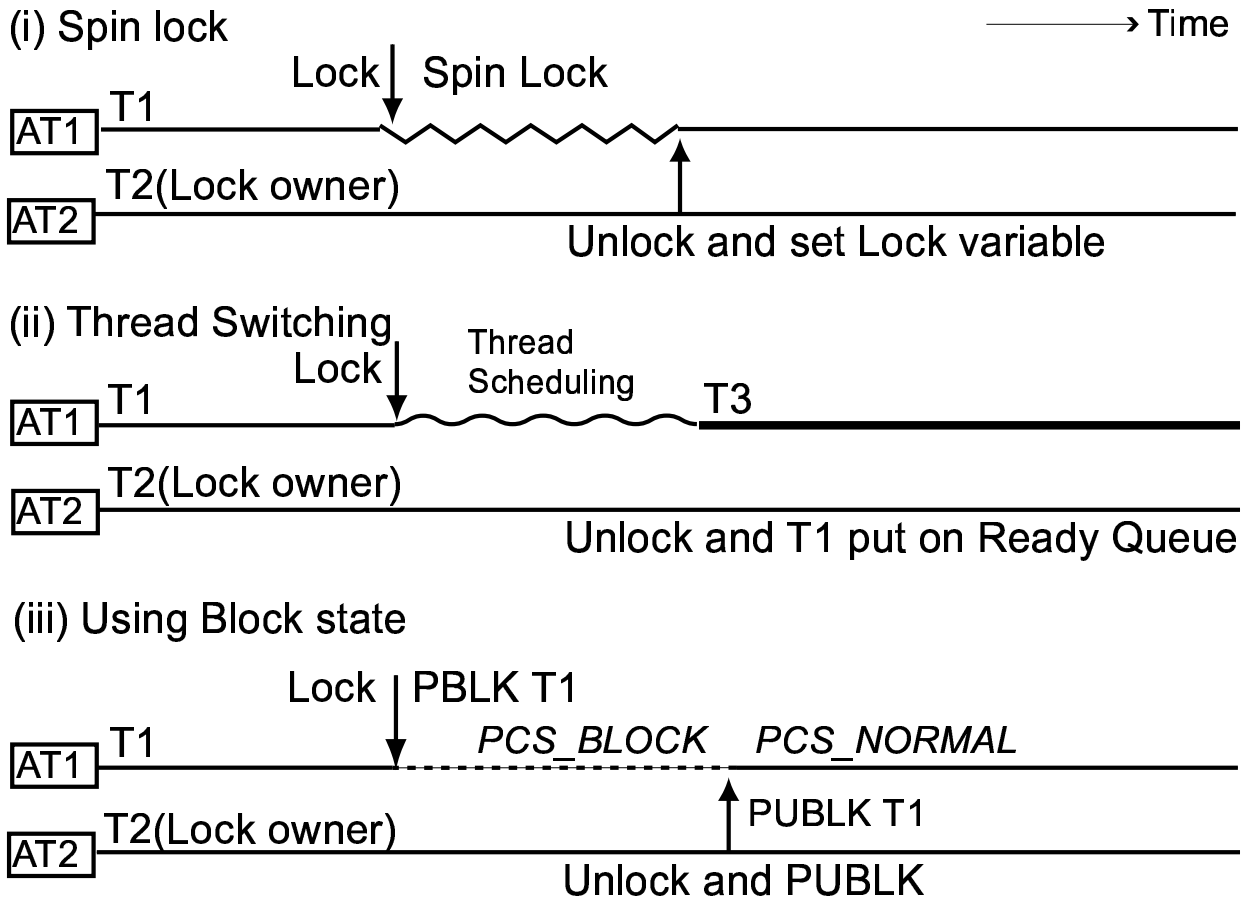

図 4.11: Mutex Lock の方式比較

従来のスレッドライブラリでは，(1) はスピンロックかスレッド切り替え，(2),(3) ではスレッド切り替えを用いて実装するのが一般的であるが，本研究ではスピンロックは行わず，スレッド切り替えも可能ならば用いない，効率のよいブロックを実現する手法を提案する．

OChiMuS PE では，実スレッドの状態を *PCS_BLOCK*，つまり一時停止状態にする命令 PBLK，それを解除する PUBLK 命令がある．MULiTh ではこれら命令を利用して，スレッドのブロックを実現する．

図 4.11 では，スレッド T1 が pthread_mutex_lock 関数によってロックを獲得しようとしたが，スレッド T2 がすでにロックを獲得しているため，スレッド T1 の実行をブロックする必要がある場合の動作を示している．図 4.11 の (i) はスピンロックを行う様子，(ii) ではスレッド切り替えによるブロックを示している．(iii) が，本研究で提案する方式である．ブロックするスレッド T1 が自分自身を対象に PBLK 命令を発行し，実スレッドの状態を *PCS_NORMAL* から *PCS_BLOCK* に遷移する．スレッド T2 がロックを解放する際，ロック解放を待っているスレッド T1 に対して PUBLK 命令を発行する．そこで，スレッド T1 は通常状態へ戻り，実行を再開する．PUBLK 命令が失敗したときには，そのスレッドはスレッド切り替えによってメモリに退避されていたことがわかり，スレッド T1 を待ちスレッドリストへ登録することにより，スレッド T1 のブロックを解除することができる．

なお，MULiTh ではブロック状態のスレッドを，ロック変数により管理するように実装した．あるロック変数について，どのスレッドがブロックしているかという情報はロック変数を参照することにより知ることができる．

本方式の利点は，スピンロックのようにループによるメモリアクセスなどを頻発せず，CPU 資源を利用しないので，他の実スレッドの動作を阻害することがない．また，PUBLK 命令一つを発行するだけですむため本方式でも高速にブロックからの復帰が行える．また，スレッド切り替え時に発生するコンテキストの保存のようなオーバーヘッドもないため高速に動作する．

提案した方式では，スレッドのブロックが発生したときに実スレッドをかならずブロックするようにすると，すべての実スレッドがブロック状態となる危険性がある．また，この方式によるブロックは実スレッドを一つ占有してしまい，スレッドの並列性を損う．そのため，レジューム可能なスレッドが存在する場合はそのスレッドへ切り替えを行うようにした．

しかし，ブロックした時点で待ちスレッドがなかったとしても，ブロックした後に他の実スレッドによってスレッド生成が起こった場合，このブロック状態の実スレッドには生成したスレッドを割り当てることができず[*5]，並列性を損なう可能性がある．

[*5] PALLC 命令が対象とするのは PCS_HALT の実スレッドのみである．

この問題点を解決するにはいくつかの方法があるが，ユーザレベルのみで対処しようとすると，大きなコストがかかることがわかった．そのため，この問題を OS との協調機構を利用して解決する．この対処については次節の OS との協調する部分で解決している．

4.5 オペレーティングシステムとスレッドライブラリの協調

MULiTh は，Future と協調動作することで効率的なスレッドスケジューリングを可能にする．この協調機構を **Kernel Notification** という．これは，カーネルで起こった事象をユーザレベルへ伝える機構を提供する．

通知する事象は次のとおりである．

- システムコール中，カーネル内でブロックする必要がある場合
- ページフォールトなど，カーネル内でブロックする必要がある場合
- ブロックしていたスレッドが再開した場合
- シグナルが発生した場合

たとえば I/O に関するシステムコールを実行中，カーネル内でそのスレッドがブロックした場合を考える．カーネルはアップコールによりユーザレベルの手続きを起動する．この手続きを **Kernel Notification Handler**(KNH) という．KNH には，ブロックしたスレッドの識別子 (LTN) と，KNH へジャンプした理由 (この場合はシステムコール中でのブロック) などが渡される．KNH では，これらの情報をもとにスレッドスケジューリングを行う．

このとき，事象が発生する前のスレッドのコンテキストをどのように KNH に渡すかが問題となる．スケジューラアクティベーション [4] ではカーネルがその領域を用意していた．また，猪原らの研究 [49] では，C-area というユーザスケジューラとカーネルの共有領域を用いて通知を最適化した．本研究では，スレッドの実行コンテキストを MULiTh が管理するそのスレッドの ThMB のコンテキスト保存領域へ直接格納する．

システムコールや例外が発生し，カーネルへ遷移する場合，カーネルはその実行コンテキストを退避する必要がある．従来の OS では，カーネルアドレス空間の領域にそのコンテキストを退避していたが，Kernel Notification ではユーザレベルの ThMB のコンテキスト退避領域にそれを格納する（図 4.12）．この方式は，実スレッドが LTN をもっており，LTN が ThMB の先頭アドレスを指すことを利用している．

従来の Scheduler Activations や C-area を利用する方式では，ThMB に相当するスレッドライブラリが管理する領域へ，ユーザスケジューラが OS から渡された実行コンテキストをコピーする必要があった．本方式では，このようなコピーが不要になるため，効率的に OS から

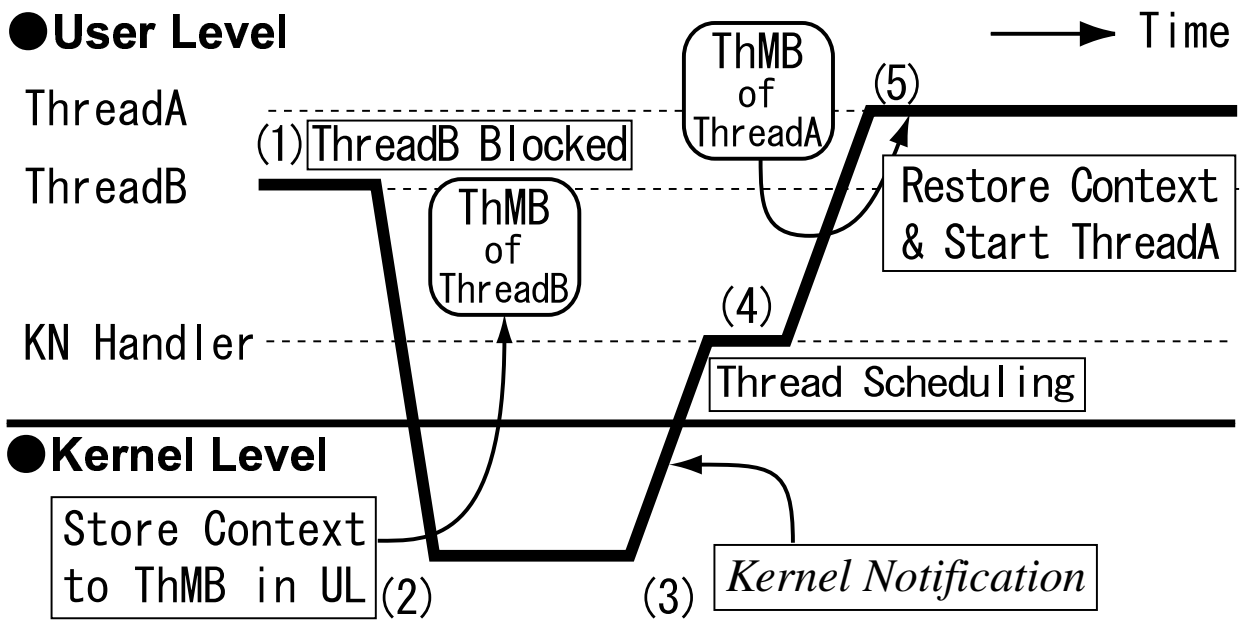

図 4.12: Kernel Notification の処理の流れ

の通知を受け取ることができる.

4.5.1 プリエンプティブなスレッド切り替え

OChiMuS PE では，プロセス切り替えはアドレス空間の遷移と，プロセッサ内のすべての実スレッドのコンテキストの退避と復帰を保証することである．Kernel Notification によるシステムでは，コンテキストの復帰は KNH で行うことでプロセス切り替えの間隔ですべての実スレッドに対し Kernel Notification が発行されることが保証される[*6].

本機構によるスレッドのプリエンプションは，通常のプロセス切り替えのコストに加え，KNH の処理，およびスレッドスケジューリングのオーバーヘッドのみであるため，軽量に実装することができる．これは，プロセス切り替えでのコンテキスト退避・復帰とスレッド切り替えのコンテキストの退避・復帰を多重化するためである.

図 4.13 では，PTL[41] などで利用しているシグナルハンドラによるスレッドのプリエンプションを示している．スレッド B を実行中，`SIGALRM` が発生し，それを機にスレッド B からスレッド A へとスレッド切り替えを行う様子を示している．図中の ThMB は，シグナルを利用するスレッドライブラリにも MULiTh におけるスレッドマネージメントブロックに相当す

[*6] 競合回避中の状態を除く.

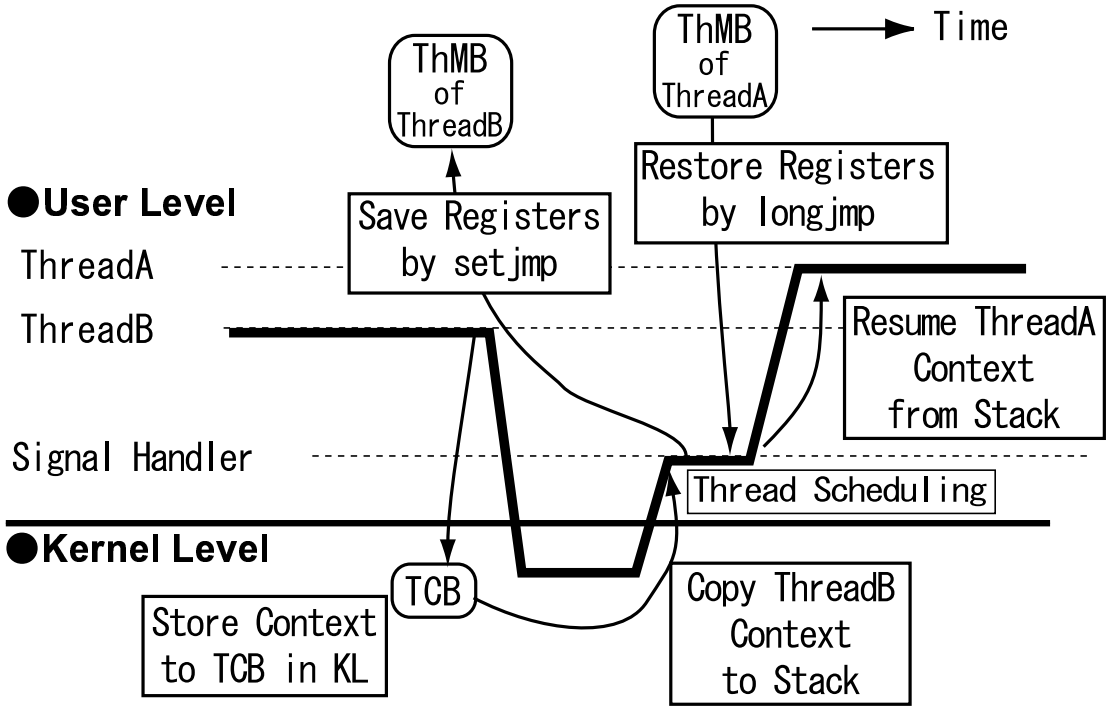

図 4.13: 従来のシグナルの利用によるプリエンプティブなスレッド切り替え

る領域と仮定しており，各スレッドがそれぞれ個別の ThMB 相当の管理領域があるとする．このシグナルによる方式は，シグナル機構が利用できればどのようなプラットホームにも移植することができるという利点はあるが，図で示しているようにコンテキストの退避，コピー，復帰が合計で 5 回ある．全レジスタのコピーが 3 回に `setjmp`，`longjmp` による復帰と退避がそれぞれ 1 回である．それに比べると Kernel Notification では実行コンテキストの復帰と退避がそれぞれ 1 回ずつしかない．コンテキストのコピーの回数以外にも，シグナルを利用する上でのカーネル内でのオーバーヘッドは大きい．

このように，Kernel Notification ではユーザレベルスレッドライブラリでの効率のよいスレッド切り替えを実現している．

4.5.2 ブロック，停止している実スレッドの扱い

前節で述べたように，プロセス切り替えが起こるとすべての実スレッドが KNH を実行する．これは，実スレッドの状態が *PCS_BLOCK*，*PCS_HALT* のスレッドも，同様に KNH へジャンプすることを保証する．

図 4.14 は，スレッド B が実スレッドを PCS_BLOCK 状態にしており，このスレッドを退避し

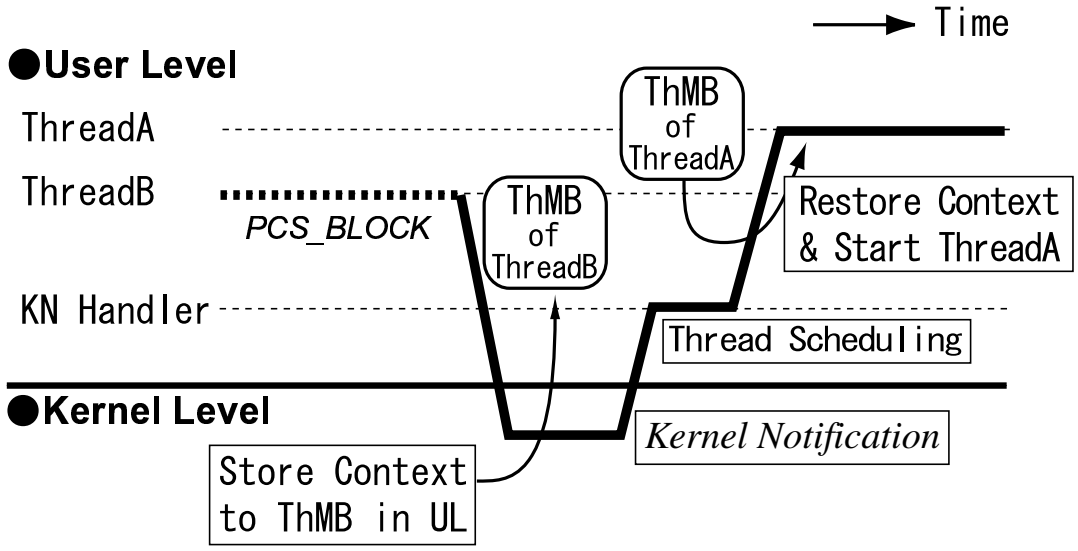

図 4.14: ブロック状態のスレッドの退避

たい場合を示している．ここで，プロセス切り替えなどがおこり，スレッド B のコンテキストはスレッド B の ThMB に退避される．実行が再開されるとき，スレッド B が動いていた実スレッドは KNH へ飛ぶ．KNH では，スレッド B を待ちスレッドリストへつながず，スレッド A へとスレッド切り替えを行う．

たとえば，スレッド A がスレッド B を起こす責任を持っていた場合，まずスレッド A はスレッド B に対して PUBLK 命令を行う．スレッド B はプロセッサ中のどの実スレッドにも存在しないため，この PUBLK 命令は失敗する．そのとき，スレッド A はスレッド B がメモリに退避されたことを知ることができ，スレッド B を待ちスレッドリストへ加える．この動作により，プロセッサのブロック状態を利用したスレッド制御が動作することが保証できる．

この機構により，プロセス切り替えのタイミングでブロック状態の実スレッドに他の待ち状態のスレッドを割り当てることができ，前述した *PCS_BLOCK* の実スレッドが長時間残ってしまうという問題点について，少なくともプロセス切り替えの間隔以上残らないように改善できる．

しかし，プロセス切り替えの間隔が長い場合にはこの問題が解決できない可能性がある．これを解決するためにはシステムコールで強制的にプロセス切り替えを発生させるなどの方式が考えられる．

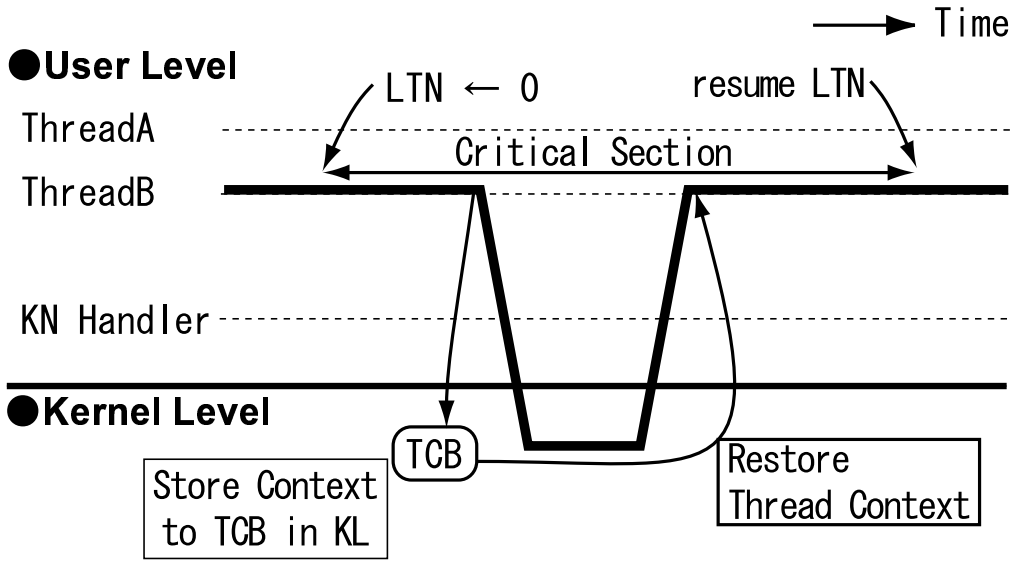

図 4.15: カーネルとスレッドライブラリとの競合の回避

4.5.3 OS との競合回避

OS とスレッドライブラリが同一の ThMB を共有するため，競合が発生する可能性がある．たとえばスレッドライブラリが ThMB に対して退避を行っているとき，例外などが発生しカーネルも ThMB へ実行コンテキストを退避すると，ユーザレベルで退避中だったコンテキストは上書きされてしまう．このようなことが起こらないよう，ThMB へのアクセスの競合を回避しなければならない．

本研究では，競合回避の方法として，ユーザレベルで ThMB にアクセスする必要がある場合，実スレッドの LTN を特別な値とすることで解決した．ここでは，その特別な値を 0 とする．

スレッドライブラリが ThMB に対してアクセスするとき，実スレッドの LTN を 0 に設定する．LTN が 0 の実スレッド上でカーネルへドメイン切り替えが起こった場合，コンテキストの退避先をカーネル内に用意した領域を利用する．また，復帰時は KNH へとぶことをせず，カーネル内に退避したコンテキストで復帰する（図 4.15）．

本手法により，スレッドライブラリと OS の ThMB への競合を防ぐことができる．

第 5 章

ソフトウェアの実装と評価

本章では実装したシステムの中でも，筆者が実装を担当したスレッドライブラリに関する部分についての実装と評価を示す.

5.1 スレッドライブラリの実装

開発環境は binutils 2.1.3 の，アセンブラを OCHIMUS プロセッサ用スレッド制御命令を扱えるように変更したものと，GCC 3.2 を用いた．C ライブラリは newlib 1.9.0 を用いた．ライブラリのコンパイルオプションは `-O3` を設定した.

スレッドライブラリ構築のために記述したソースコードは，合計 10 ファイルで約 2000 行であった．ライブラリの中では，プロセッサのスレッド制御命令を利用するため，GCC の拡張アセンブラ表記を利用した MIPS アセンブラの記述を約 40 個所で行った.

5.2 評価環境

本評価は筆者らが作成している実行駆動型シミュレータ **MUTHASI**(MUltiTHreaded Architecture SImulator)[44] を用いて行った．MUTHASI は OChiMuS PE をシミュレートし，プロセッサのパラメータについて容易に設定可能となっている．また，実スレッド数を 1 に設定すると，通常の MIPS プロセッサの挙動を示す．なお，本論文ではキャッシュを機能せずに評価を行った.

プログラムの作成は binutils-2.13[*1]，gcc-3.2[*2]，newlib1.9.0 を用いた．MULiTh 自体の機

[*1] OChiMuS PE のスレッド制御命令を利用可能にしたもの.

[*2] 最適化オプションは -O3 を設定した.

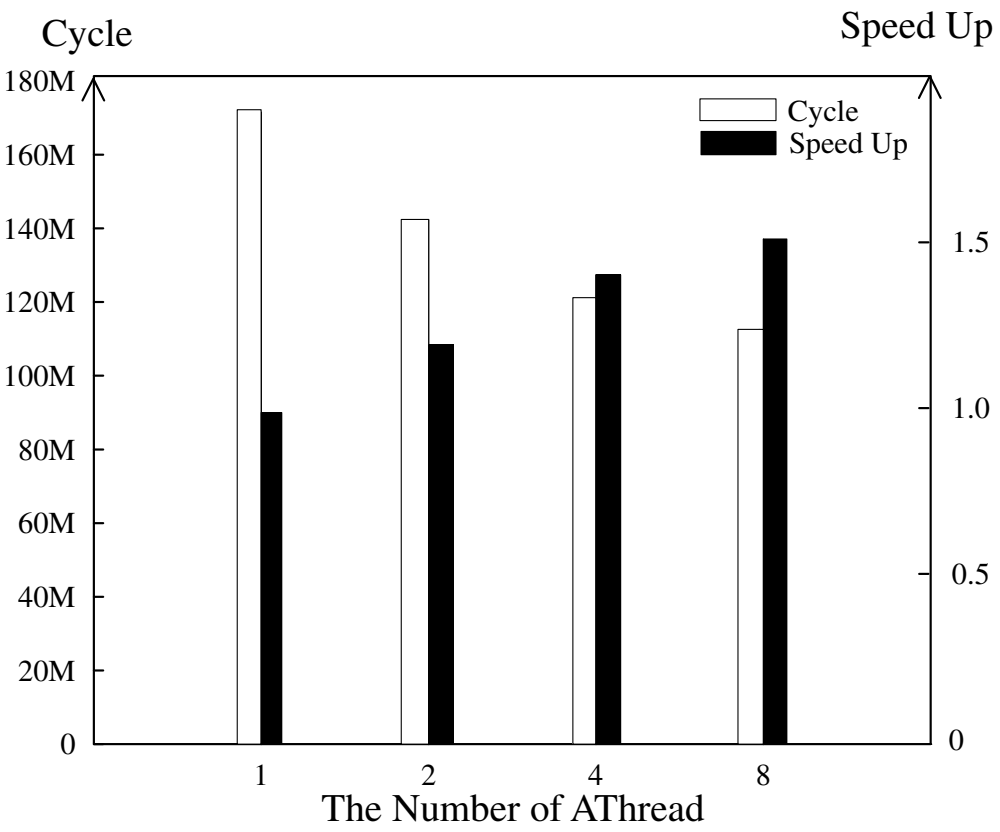

図 5.1: 並列実行した結果

能はスレッドの生成，削除，同期，排他制御の基本動作について実装した．スレッド制御の詳細な設定については現状ではサポートしていない．

5.3　並列化したプログラムの速度向上率の評価

図 5.1 は，平均画素法による画像縮小プログラムを MULiTh の Pthread インターフェースを利用してマルチスレッド化したプログラムの実行結果である．実スレッドの数が 1 の場合，従来のスーパースカラアーキテクチャプロセッサ，それ以外がマルチスレッドアーキテクチャによる並列実行した結果となっている．

スレッドの並列実行によって，処理速度を向上させることができたことが確認できる．

表 5.1: スレッド制御の性能

評価項目	(1) 提案方式	(2) 従来方式または (1) が利用できない場合 (*)
スレッド生成	84 (74)	135 (102)
スレッド削除	51 (42)	223 (126)
同期	202 (95)	847 (515)
	(1) 提案方式	(2)Spin Lock
排他制御 (短時間)	972	982
排他制御 (長時間)	41461	46656
	(1) 提案方式	(2) 従来方式
プリエンプション	523 (341)	878 (718)
OS からの通知	373 (256)	522 (418)

* スレッド制御命令が失敗した場合，または利用できない場合

** 単位は総実行サイクル数 (かっこ内は実行命令数)

5.4 スレッド制御の評価

5.4.1 スレッドライブラリの評価

表 5.1 は，スレッドライブラリの各操作の性能を測定した結果である．シミュレータの実スレッド数を 4 にして評価を行った．以下，各測定項目について述べる．

評価に利用したシグナル機構，Scheduler Activations 機構については，カーネルからの通知機構部分のみを実装した．

- スレッドの生成

  (1) は PALLC 命令が成功した場合, (2) は従来のスレッドライブラリの動作，または PALLC 命令が失敗したときの動作である．(2) では実際に生成したスレッドが動作するには，加えてスレッド切り替えのオーバーヘッドがかかる．PALLC 命令が成功して新たなスレッドが実スレッドとして割り当てられたとき，処理の多くがレジスタアクセスのみですむた

め，高速にスレッドを生成できることがわかる．

- スレッドの削除

  (1) は，待ちスレッドがない場合の PDALL 命令によるスレッド削除，(2) は従来のスレッド削除における他のスレッドへ切り替え，または MULiTh においてのスレッド切り替えを示している．PDALL を実行する (1) では，スレッドの削除が 1 命令ですむので非常に軽量である．

- 同期

  (1) はプロセッサのブロック状態を利用した方式，(2) は従来のスレッド切り替えによる方式の結果である．また，MULiTh でも，待ちスレッドがある場合は (2) の動作を行う．提案方式では，コンテキストの退避と復帰を行わないので，スレッド切り替えを行う場合に比べて 4 倍の性能向上となった．

- 排他制御

  ロック獲得のための待ちに，(1) はプロセッサのブロック状態を利用した方式，(2) はスピンロックを利用した方式である．短時間，長時間とは，クリティカルセクションの長さである．短時間の処理では，違いはあまり見られない．しかし，長時間排他制御による保護を行うと，スピンロックの利用は性能が落ちている．これは，スピンロックによって，他の実スレッドの実行を妨げているためであり，提案する方式の有用性が確認できた．

- プリエンプション

  (1) は提案する Kernel Notification による非同期のスレッド切り替え，(2) は UNIX のシグナル通知によるプリエンプションである．提案する Kernel Notification では，従来のシグナル機構における何度かのコンテキストのコピーが行われないため，プリエンプションが効率的に行われている．実際のシステムでは，プロセス切り替えとスレッド切り替えのコンテキストの退避・復帰を多重化するため，より高性能となる．

- OS からの通知

  (1) は提案する Kernel Notification による OS からの通知，(2) は Scheduler Activations 機構，特にそれを効率的に行う C-area[49] による通知と同等の処理を実装し，比較した．Kernel Notification ではスケジューラ内でのコンテキストのコピーが行われないため，高速に実行することができている．特にメモリアクセスが少なくなるため，キャッシュを有効にした際には効果があると考えられる．

表 5.2: Performance of Fine grained thread creation

	Execution insns
Normal thread creation	102
Report failure	62
Report failure and cache LTN	26

Unit is instruction number.

表 5.3: fib(25) の実行速度

	Execution time	Speedup ratio
Serial Execution	2522604	1.00
With normal MULiTh	10698075	0.25
Using support of fine-grain thread creation	2032359	1.24

単位は実行にかかった総サイクル数.

5.5 細粒度スレッド生成の評価

表 5.2 は細粒度スレッド生成を利用した場合の命令数を計測した結果である．通常のスレッド生成プロセスでは実スレッド割り当てが失敗した場合の処理に 102 命令かかる．しかし提案手法により，このオーバヘッドを 26 命令に抑えることに成功した.

図 5.2 は，N 番目のフィボナッチ数を求めるプログラムである（図中の pthread_create_fg が細粒度スレッド生成をサポートする関数である）．このプログラムで 25 番目のフィボナッチ数を求めたときの実行結果を，逐次実行したものと通常の MULiTh を利用したものと，細粒度スレッド生成サポートを利用した場合の比較を行った (表 5.3)．この結果，通常の MULiTh ではスレッド生成のオーバヘッドにより逐次実行よりも 4 倍近く遅くなっているが細粒度スレッド生成サポートを利用すると 1.24 倍の性能向上を実現することができた．これは、通常のスレッド生成プロセスを利用した場合に比べて実に 5 倍の性能向上を示すものである.

```
void *fib_th(void *p){
  return (void *)fib((int)p);
}
int fib(int n){
  if(n <= 2)
    return 1;
  else{
    pthread_t t1;
    int a1 = 0, a2 = 0, err;
    err = th_create_fg(&t1, 0,
                       fib_th, (void*)n-1);
    if(err == E_AGAIN)
      a1 = fib(n-1);
    a2 = fib(n-2);
    if(a1 == 0)
      pthread_join(t1,(void*)&a1);
    return a1+a2;
  }
}
```

図 5.2: N 番目のフィボナッチ数を求めるプログラム

5.6 議論

5.6.1 本方式の一般性

本研究で実装し評価したスレッドライブラリなどのシステムソフトウェアは OChiMuS PE プロセッサを対象として議論を行ったが，本研究での提案はこのプロセッサアーキテクチャのみに限定されるわけではない．たとえば OChiMuS PE はベースとして MIPS アーキテクチャを採用していたが，実スレッドの制御が OChiMuS PE と同様に可能であれば，ベースとなるアーキテクチャは問わない．たとえば Intel の IA32 アーキテクチャである Xeon プロセッサが実スレッド管理をユーザレベルで OChiMuS PE のように制御可能であれば，本研究の提案を適用することができる．

また，CMP，オンチップマルチプロセッサでも，同様の管理機構がプロセッサアーキテクチャとして提供されていれば本研究が適用可能である．これは，マルチスレッドアーキテ

クチャ特有の問題点や課題，たとえばメモリポートの競合が起こると全体性能の劣化が起こる，などが SMT アーキテクチャと共通の問題としてあるからである．しかし，計算モデルは CMP ，SMT とそれぞれ別のものになることが考えられる．これは，本論文での評価で行ったような行列演算の掛け算のような，SMT アーキテクチャでは演算器の競合によって性能が向上しなかった計算に対し，CMP では処理性能の向上が期待できるという点である．SMT アーキテクチャと違い，CMP ではそれぞれの実スレッドにあたるプロセッサが個別に演算器を所有しているため，演算器の競合が起こらないためである．しかし，逆に演算器の共有によって処理速度が向上した画像縮小の例のような計算では，CMP では逆に演算器に無駄が出る可能性がある．このように，得意な計算モデルに違いはあるが，スレッドライブラリの構築の点から見ると，本研究は適用可能であると考えられる．

OS アーキテクチャは，現在 OS Future を対象にスレッドライブラリと協調動作を行っているが，Linux などの他の OS にもマルチスレッドアーキテクチャ向けのプロセス管理を実装し，実現することは可能である．また，Kernel Notification の機構はマルチスレッドアーキテクチャのみでなく従来の SMP 計算機などでも，物理プロセッサを実スレッドとして扱うことで，Kernel Notification などの考え方を応用可能ではないかと考えている．

第 6 章

結論

6.1 成果

6.2 今後の課題

スレッドライブラリ MULiTh は，まだ Pthread 仕様に関して未実装の部分がある．たとえばスレッドのキャンセルや優先度付きスケジューラなどは実装していない．未実装部分の着手を行う予定である．

での提案はこのアーキテクチャだけに限定されるものではない．たとえば OChiMuS PE では MIPS アーキテクチャを採用していたが，実スレッドの制御を可能にするアーキテクチャであれば，ベースとなるアーキテクチャは問わない．たとえば Xeon プロセッサが実スレッド管理制御をユーザレベルで可能にすれば，本研究での提案を適用することができる．また，OS は今後 Linux などにマルチスレッドアーキテクチャ向けプロセス管理を実装し，MULiTh と協調動作の実現させる予定である．

本提案は，Future と連携し，実用的なアプリケーションを動作させることで，効果を発揮することが期待できる．今後，キャッシュを搭載したシミュレータや，シミュレータではなく実際にチップとして動作する OChiMuS PE での評価など，より現実的な環境による評価を通じてその可能性を明らかにしていきたい．そして，Future の OS アーキテクチャを Linux などに移植し，スレッドライブラリ MULiTh と連携動作させることでより現実的な評価を行いたい．

謝辞

本研究を進めるにあたり，筆者が所属する並木研究室の方々，および中條研究室の方々には日ごろからいろいろな助言やご指導を頂きました．また，学内の各先生，各学生諸子にもご指導頂くなどお世話になりました．学外でも，他大学はもとより様々な方にお世話になりました．心より感謝いたします．

東京農工大学工学部情報コミュニケーション工学科並木研究室学部 4 年の内倉 要氏，小高 健二氏，古川 政嗣氏，森 拓郎氏，シャロー イップ氏，博士前期課程 1 年の下屋鋪 太一氏にはいろいろと筆者の足りない部分について気づかせていただきました．感謝します．本学並木研究室博士前期課程 2 年の大塚 雄三氏，林寛氏，増渕 敬氏，益子 由裕，西本 聡氏には研究室の先輩として数々のアドバイスをいただきました．またゼミや中間発表において有益なアドバイスを頂きました．感謝いたします．研究室生活を楽しく過ごせたのは，研究室の皆様のおかげです．

本研究は本学並木研究室，中條研究室と合同で研究を進めている OChiMuS プロジェクトの一部です．関係者各位に感謝いたします．

本学科中條研究室学部 4 年の辻本 治氏にはシミュレータ MUTHASI のメンテナンスに大いに貢献していただきました．感謝いたします．同博士前期課程 1 年の加藤義人氏，大和 仁典氏には，研究についてミーティングなどで相談に乗っていただき，また日ごろの生活でもいろいろとお世話になりました．感謝いたします．NEC シリコンシステム研究所の河原 章二氏には，本研究の礎を築いていただき，また折々に必要な助言を頂きました．深く感謝いたします．

有志で開催している SICP 読書会の参加者の皆様，RHG 読書会の参加者の皆様にはプログラミング関連の様々な話題について実に刺激的な時間を過ごさせて頂きました．また．また，本当にいろいろな示唆を頂きました．深く感謝いたします．

ネットワーク応用通信研究所の松本 行弘氏はプログラミング言語 Ruby の開発者として，現在もメンテナンス，および開発を継続されています．Ruby は筆者をその処理系（Ruby 仮

想マシン）の開発に夢中にしてくれました．また，VM 作りを行おうか迷っている筆者の背中を押していただきました．深く感謝いたします．

筑波大学の前田 敦司助教授，産業技術総合研究所の首藤 一幸氏にはまったくの素人である筆者に言語処理系，主にインタプリタや仮想マシンについての多くの助言を頂きました．深く感謝いたします．

法政大学の中田 育男教授，および COINS プロジェクトの皆様にはコンパイラを中心とした言語処理系開発における様々なことがらについて勉強させていただきました．深く感謝いたします．

本学並木研究室博士後期課程 2 年の佐藤未来子氏には，Future OS の作成から，システムソフトウェアについての開発で，お忙しい中，日ごろから多くのご指導をいただきました．また，筆者の論文のチェックなどもしていただきました．深く感謝いたします．

中條 拓伯助教授には，日ごろからご指導いただいきました．深く感謝いたします．

並木 美太郎助教授にはこの 2 年間，多くのご指導をいただきました．日ごろのゼミでの専門の立場からのご指導，研究室生活のバックアップをしていただいたり，発表のチャンスを頂いたりと，本当にお世話になりました．深く感謝いたします．

最後に，不規則な生活の筆者を支えてくれた家族に，深く感謝いたします．

参考文献

[1] Java technology. http://java.sun.com/.

[2] Openmp: Simple, portable, scalable smp programming. http://www.openmp.org/.

[3] Python programming language. http://www.python.org/.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79, 1992.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[6] Intel Corporation. *Intel(R) Pentium(R) 4 Processor - Product Overview.*

[7] Intel Corporation. IA-32 インテル (R) アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻：命令セット・リファレンス, 2004.

[8] Ulrich Drepper and Ingo Molnar. The new native posix thread library for linux : Nptl. http:// people.redhat.com/ drepper/ nptl-design.pdf : Draft.

[9] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, Vol. 17, No. 5, pp. 12–19, 1997.

[10] Jason Evans. Kernel-scheduled entities for freebsd. http://www.aims.net.au/chris/kse/tex/, 2 2003.

[11] Manu Gulati and Nader Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, p. 291. IEEE Computer Society, 1996.

[12] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio

Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th annual international symposium on Computer architecture*, pp. 136–145. ACM Press, 1992.

[13] IBM. Next generation posix threading : Ngpt. http://www-124.ibm.com /developerworks/ opensource/ pthreads/.

[14] IEEE. *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1*, 1996.

[15] Intel. Intel technology journal(hyper-threading technolog. http://www.intel.co.jp /jp/developer /technology/itj/.

[16] Steve Kleiman, Devang Shah, Bart Smaalders, 岩本信一訳. 実践マルチスレッドプログラミング. サンソフトプレスシリーズ. 株式会社アスキー, 1998.

[17] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 39–50. IEEE Computer Society, 1998.

[18] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 114–124. IEEE Computer Society, 1997.

[19] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp. 322–354, 1997.

[20] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the13th ACM Symposium on Operating Systems Principle*, pp. 110–121, Pacific Grove, CA, 1991.

[21] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors, 2000.

[22] David A. Patterson, John L. Hennessy, 富田真治訳, 新実治男訳, 村上和彰訳. コンピュータ・アーキテクチャ—設計・実現・評価の定量的アプローチ. 日経 BP 社, 1994.

[23] Diego Puppin and Dean Tullsen. Maximizing tlp with loop-parallelization on smt. In *Workshop on Multithreaded Execution, Architecture and Compilation*, 2001.

[24] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. *SIGPLAN Not.*, Vol. 35, No. 11, pp. 245–256, 2000.

[25] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proc. 2nd Annual ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 70–78, Seattle, WA (USA), 1990.

[26] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pp. 234–244, 2000.

[27] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 66–76. ACM Press, 2002.

[28] Dan Stein and Devang Shah. Implementing lightweight threads. In *Proceedings of theSummer 1992 USENIX Technical Conference and Exhibition*, pp. 1–10, San Antonio, TX, 1992. USENIX.

[29] Dominic Sweetman. *See Mips Run.* Morgan Kaufmann Pub, 1999.

[30] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 60–71. ACM Press, 1999.

[31] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. In *In Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, No. 1107 in Lecture Notes in Computer Science, pp. 59–82, 1996.

[32] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the unix kernel: The battle for control. Technical Report CMU-CS-87-149, 1987.

[33] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.

[34] Xavier.Leroy. The linuxthreads library. http://pauillac.inria.fr/ xleroy/linuxthreads/.

[35] K. Yoshizoe, Matsumoto, T., and K. Hiraki. Speculative parallel execution on jvm. In

First UK Workshop Java for High Performance Network Computing at EuroPar'98, 1998.

[36] ティム・リンドホルム, フランク・イェリン. Java 仮想マシン仕様第 2 版. ピアソン・エデュケーション, 2001.

[37] 美添一樹, 松本尚, 平木敬. ループ並列投機実行の java 仮想マシンへの適用. 情報処理学会ハイパフォーマンスコンピューティング研究会研究報告, No. 72, pp. 1–6, 1998.

[38] 大河原英喜, 安里彰. 最適実行多重度に基づく smt プロセッサのジョブスケジューリング方式. 情報処理学会 研究報告「計算機アーキテクチャ」, No. 150, 2002.

[39] 多田好克, 寺田実. 移植性・拡張性に優れた C のコルーチンライブラリー実現法. 電子情報通信学会論文誌, Vol. J73D-I, No. 12, pp. 961–970, 1990.

[40] 安倍広多, 松浦敏雄, 安本慶一, 東野輝夫. ユーザレベル軽量プロセスライブラリにおける効率の良い I/O 処理方式. 情処研報 98-OS-79, Vol. 98, No. 71, pp. 77–84, 1998.

[41] 安倍広多, 松浦敏雄, 谷口健一. BSD UNIX 上での移植性に優れた軽量プロセス機構の実現. 情報処理学会論文誌, Vol. 36, No. 2, pp. 296–303, 1995.

[42] 福田晃. 並列オペレーティングシステム. コロナ社, 1997.

[43] 岡坂史紀, 清水謙多郎, 芦原評, 亀田壽夫. ユーザプログラムとカーネルの協調に基づくスレッドの設計と実現. 情報処理学会論文誌, Vol. 36, No. 4, pp. 913–924, 1995.

[44] 河原章二, 佐藤未来子, 並木美太郎, 中條拓伯. システムソフトウェアとの協調を目指すオンチップマルチスレッドアーキテクチャの構想. コンピュータシステムシンポジウム, Vol. 2002, No. 18, pp. 1–8, 2002.

[45] 近藤拓也, 日下部茂. アドレス空間を共有するプロセスの集約を行う定数オーダースケジューリング. SACSIS2003 予稿集, pp. 21–24, 2003.

[46] 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一. オブジェクト指向並列言語 opa のためのコード生成手法. 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG11, pp. 1–13, Nov 2001.

[47] 小池汎平, 山名早人, 山口喜教. 逐次プログラムの投機並列実行を行う中間コードインタプリタの構成法. 情報処理学会論文誌：プログラミング, Vol. 40, No. SIG 10(PRO 5), pp. 64–74, Dec 1999.

[48] 佐藤未来子, 河原章二, 中條拓伯, 並木美太郎. SOC 時代に向けた SMT 用 OS の構想. システムソフトウェアとオペレーティング・システム, No. 91-5, pp. 31–38, 2002.

[49] 猪原茂和, 益田隆司. ユーザとカーネルの非同期的な協調機構によるスレッド切り替え動作の最適化. 情報処理学会論文誌, Vol. 36, No. 10, pp. 2498–2510, 1995.