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

Implementation and Evaluation of a Thread Library for Multithreaded Architecture

笹田　耕一

Koichi Sasada

(1999 年度入学, 99249031)

指導教官　並木 美太郎 助教授

東京農工大学 工学部 情報コミュニケーション工学科 SC コース

2002 年度卒業論文

(2003 年 1 月 31 日提出)

目次

図目次

表目次

第 1 章

緒言

本研究では，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，および評価を行った．本章ではその背景と目的について述べる．

1.1 背景

本研究の背景を述べる．

1.1.1 スレッドプログラミング

今日，スレッドプログラミングは非常に重要なプログラミングテクニックとして知られており，サーバプログラミング，GUI(Graphical User Interface) プログラミング，並列分散処理などに広く利用されている．これは，プログラミングの記述性の問題から，性能，レスポンスタイムの向上を目指すものまでさまざまな理由がある．

スレッドとは何か，という定義について，UNIX 流のプロセスと並列処理を契機として再認識されたスレッドの概念を改めて述べる．従来のオペレーティングシステム (以下，OS) はユーザに計算ハードウェアを仮想化した計算機環境を提供しており，OS がユーザプログラムを実行することになる．このとき，OS はユーザプログラム実行のためのなんらかの抽象化と機構を提供する必要がある．この抽象化されたものがプロセスであり，仮想的なプロセッサとして捉えらる．つまり，物理プロセッサを仮想化したものであり，仮想化されたメモリなどとともにプログラムを実行する環境を提供する，すなわちプロセスはプログラムのための器であると考えられる．

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

表 1.1: 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

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 1.1 のようになる [22].

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．[22]

このような利点をもつスレッドを利用することで，次のような効用を得ることができる．

遅延隠蔽　一般的にプログラムがある仕事を実行するときには，その仕事の実行を妨げるさまざまな遅延が生じる．これは並列プログラムだけでなく，逐次プログラムにおいても，入出力を行うことで生じる．このとき，一つの CPU あたり複数のスレッドを持つことにより，CPU 時間を無駄にすることなく他のスレッドを実行することができる．このため，CPU の利用効率を向上させ，全体の処理性能をあげることができる．

記述性　平行して実行させたい複数の仕事を複数のスレッドに行わせることで，プログラムの記述性を向上させることができる．たとえば，GUI アプリケーションでは，ある描画処理を行いながら外部からの入力に常に反応することを要求される場合がある．また，Web サーバでは，あるクライアントからのリクエストに対応しながらも他のクライアントの接続や要求を受け付けなければならない．このようなプログラムを従来のシングルスレッドモデルで記述しようとす

ると非常にプログラムが複雑になる．複数のスレッドを用いることで，各々の処理の記述は逐次プログラム同様の単純さを保ったまま，目的を達成することができる．

負荷分散　複数の CPU を持つシステムなどでは，スレッドを自然な負荷分散の単位とすることができる．複数のスレッドが並列に仕事を行うことにより，全体の処理時間を短くすることができる．このとき，スレッドの数を CPU の数よりも多くすると，スレッドごとの仕事量の違いによる負荷の偏りを緩和しやすくなる．

マルチスレッド化に応じてもっとも性能向上に有効となるのは負荷分散である．負荷分散を行うためには，スレッド制御，とくに生成と削除が軽量に行うことが要求される．これは，全体の仕事量におけるスレッド制御のオーバーヘッドが大きければ，性能向上が見込めないからである．

1.1.2　マルチスレッドプロセッサアーキテクチャ

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) に着目してきたものが主流であった．その中で，アウトオブオーダー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流の中から並列性を検出し，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 [16] 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．

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 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終章で述べる．

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．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．

たとえば，複数のプロセッサをもつ SMP(Symmetric MultiProcessor) 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 [25]．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．

1.2 目的

本研究ではマルチスレッドアーキテクチャにおける高性能なスレッドライブラリの作成を目的とする．従来のシステムソフトウェアの枠組みのもとで，従来のスレッド処理機構の手法をそのまま適用すると，マルチスレッドアーキテクチャの利点が生かすことができない．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．

処理性能を向上させるためには，軽量なスレッド制御により処理の並列実行を行いたい．そのため，本研究ではマルチスレッドアーキテクチャプロセッサが複数の実スレッドを持ち，それを並列に実行できるという特性を活用することで処理速度を向上させることを目指す．そのためにはスレッドライブラリをどのようなモデルで作成するか，という議論が必要であり，これについては次章で従来のモデルを考察し，マルチスレッドアーキテクチャに適したモデルを考察する．また，実スレッドの管理方式を従来のシステムソフトウェアの方式を見直すことで再度検討する．4 章で考察したモデルを実現するためのシステムの全体像を示し，これを実現するプロセッサ，OS アーキテクチャに

ついて述べる．本研究で提案するスレッドライブラリがどのようにマルチスレッドアーキテクチャプロセッサを活用するか，という具体的な方式設計は 5 章で行う．

C 言語でのプログラムが作成できる環境で利用可能なスレッドライブラリであれば，ソースレベルの互換性を提供することができ，すでに開発されている豊富なプログラムを利用することができる．そのため，新たな言語処理系の開発などのはせず，C 言語の枠組みで利用できるスレッドライブラリの開発を目指す．一般的に多くの言語処理系では C 言語用ライブラリを利用するための手段を有しており，その点を考えてもこのアプローチは汎用的であるといえる．このライブラリのプログラミングインターフェースの詳細については 5 章で述べる．

本研究では性能向上を第一の目標とするため，ユーザレベルで実行可能なスレッドライブラリを目指す．ユーザレベルでスレッドを管理する際に生じる困難な問題については OS と協調してこれを解決する．この検討を 5 章で述べる．

また，本研究の有用性を示すために実装し評価を行う．これについては 6 章で論じる．

第 2 章

問題分析

スレッドを利用するには，スレッドをサポートするためのシステムが必要になる．本章では，既存のスレッド処理機構の実装方法についてまとめる．また，従来のシステムソフトウェアでマルチスレッドアーキテクチャを利用するとき，何が問題となるかを述べ，それについて考察する．

2.1 従来のスレッド処理機構

スレッドを利用可能にするためには，それを実現するシステムが必要になる．そのシステムは一般的には，ユーザーレベルでのライブラリによる実現，カーネルレベルでのオペレーティングによるサポート，それら二つを組み合わせた方法，または言語処理系（解釈実行系による実装やコンパイラによる並列化コードの生成）によるサポートなどがある．

具体的に，UNIX などのシステムを考えると，プログラムが実行するとき，次の二つのモードがある．

- カーネルレベル
- ユーザレベル

カーネルレベルでは，すべてのハードウェア資源などに直接アクセスができるレベルで，OS はこのレベルで動作する．ユーザレベルとは，一般的なハードウェア資源などは OS にリクエストを発行することで利用でき，各プロセスで独立な仮想メモリを持つ．一般的に，ユーザレベルからカーネルレベルへのリクエストはシステムコールなどを用いて行い，それにともなってコンテキスト退避やプロセッサのモード切替えなど，オーバーヘッドが生じる．

スレッドをどのレベルで実現するか，という点で，カーネルレベルスレッド，ユーザレベルスレッドという概念が生まれた．また，そのどちらも利用する 2 レベルスレッド，協調型スレッド処理機構という方法も提案されている．本節ではこれ以降，それぞれのスレッドモデルについて述べ，それらの長所と短所についてまとめる．

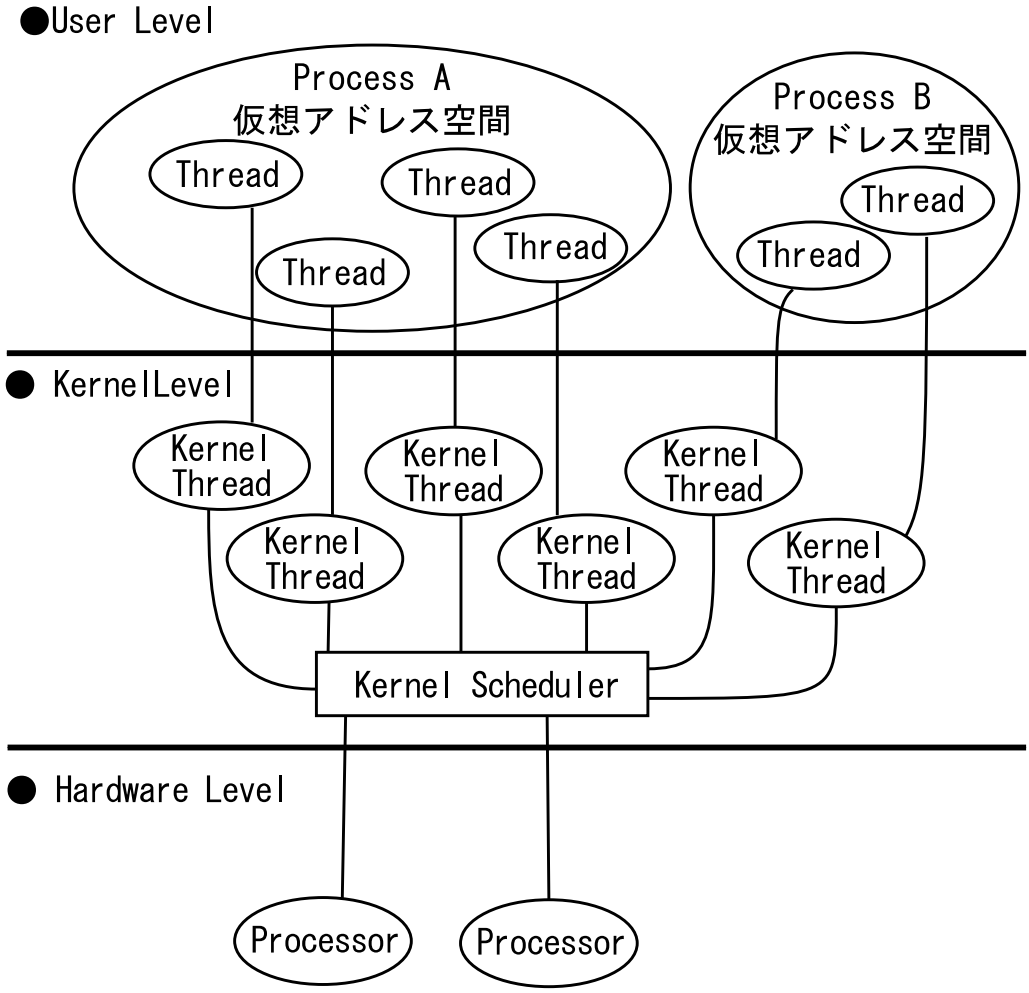


図 2.1: カーネルレベルスレッドモデル

2.1.1 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 2.1）．図中ではプロセス A がスレッドを 4 つ有しており，カーネル内にも対応するかーねるすれっどが 4 つ用意されている．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[15] などが有名であるが，Linux での Linux Thread[18] や，論文執筆時 (2003 年 1 月) での次期バージョンの Linux 用スレッドライブラリといわれている NPTL[3] 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 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レッド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を行うことができない．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．

2.1.2 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 2.2）．プロセス生成時に一つカーネルスレッドが生成され，そのカーネルスレッドに対してユーザレベルで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 [19] 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 [21] では，シグナルを用いた移植性の高いプリエンプティブなコンテキスト切り換えを実現している．

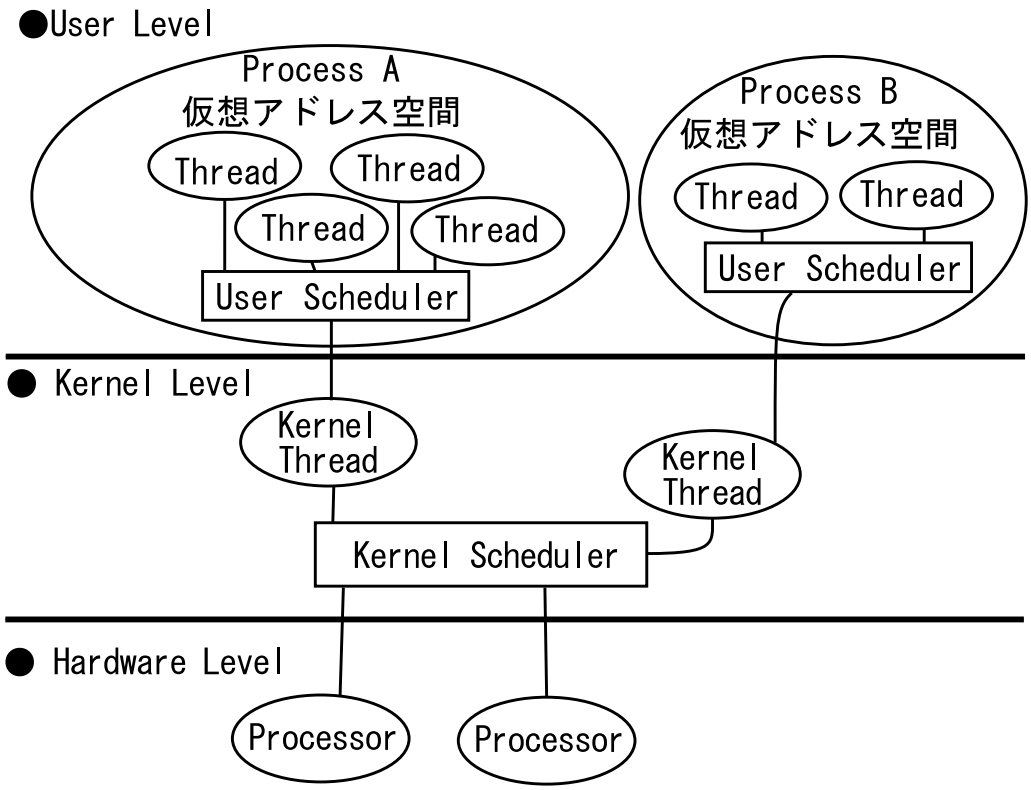


図 2.2: ユーザレベルスレッドモデル

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点があげられる．また，特定の OS に依存しない方法をとることができるため，移植性の高いライブラリを作成することが可能である．

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．

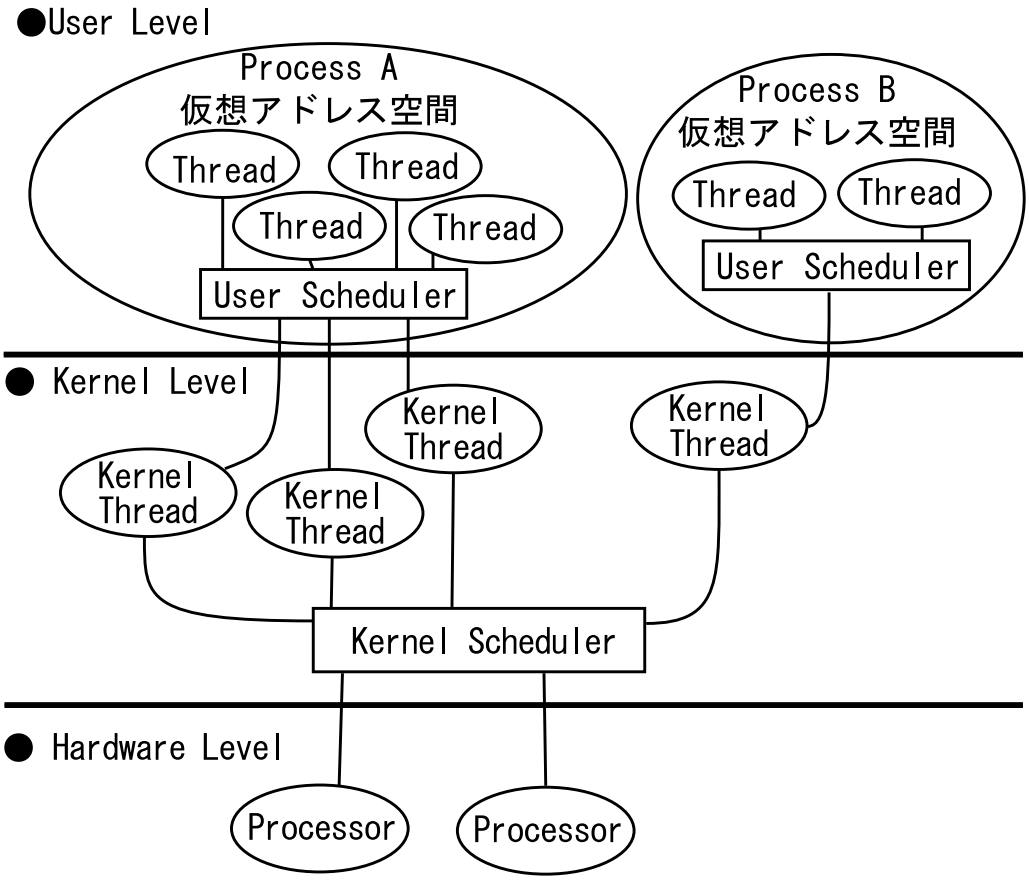


図 2.3: 2 レベルスレッドモデル

2.1.3 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 2.3）．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 $N \leq M$ であり，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 [26] 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[4] ，安部らの I/O ブロッキング回避たのめ，カーネルスレッドを生成するために他プロセスを生成する研究 [20] 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．

2.1.4 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による情報の流れの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，その逆の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 [1]．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

表 2.1: 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	○	○	○	×

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[10] 上に実現されている [8]．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 [26] 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 [23]．商用 OS としては SunOS5.0[12] などがある．

2.1.5 スレッドモデルについての考察

表 2.1 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．

本研究は高性能なスレッドライブラリを追求することが目的なので，協調型スレッドモデルを採用したい．これを実際に実現する方法については 4 章と 5 章で論じる．

2.2 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

マルチスレッドアーキテクチャプロセッサにおけるスレッドライブラリを構築するときに問題となる点について示す．

2.2.1 従来の実スレッド管理方式

マルチスレッドアーキテクチャプロセッサは，複数の実スレッドにより実行命令流を並列に動作させることが可能なプロセッサである．しかし，どのような実行命令流を実行するか，というのは

プロセッサを利用するソフトウェアが決めなければならない.

その単位として，たとえば次のようなものが考えられる．

1. 並列化コンパイラが生成した並列実行可能な命令流（細粒度）
2. スレッド（中粒度〜粗粒度）
3. プロセス（粗粒度）

並列化コンパイラによって得られた命令流は，一般的に細粒度である．これは，ループを展開したものなどがあげられる．プロセスの生成には一般的にコストが大きいため，粗粒度の命令流となる．スレッドは，スレッドモデルによるが，ユーザレベルで管理する場合，生成コストが低いため中流度から粗粒度を扱う命令流となる．

細粒度の処理を行うため, 遅延実行 (Lazy Task Creation)[9] を利用した並列化言語処理系 [13][17] や，既存の言語処理系を拡張したもの [2][14] を用意するなどが考えられるが，本研究が対象とするのは言語処理系に対して特別な改変を必要とせず，より一般的な機構，つまりスレッドライブラリによる汎用的なスレッド機構を提供することにある．そのため，コンパイラや言語処理系を利用する並列化については対象としない.

本項では，中粒度〜粗粒度，つまりスレッドやプロセスについて，従来のシステムソフトウェアがどのように管理しているかについて述べ，その問題点を示す.

たとえば，現在製品として出荷されている SMT アーキテクチャのプロセッサである Xeon プロセッサ [6] は，1 プロセッサに 2 個のプログラムカウンタを持ち，仮想的に 2 つの CPU を持つ SMP 計算機として動作する．仮想的な SMP 計算機として動作するため，従来の SMP 計算機向けシステムソフトウェアをそのまま利用することができるという利点がある.

しかし，この方法では，一つのプログラムカウンタが 1 個のプロセスを割り当てられることになり，1 プロセッサで扱うワーキングセットが増大し，キャッシュや TLB ミスが多発する．図 2.4 は，シングルスレッドのプロセス A，プロセス B をプロセッサに割り付け，Xeon プロセッサ内の二つの実スレッドが同時にプロセス A，プロセス B を実行する様子を示す．アドレス空間が別々のプロセス A，B を同時に処理するためには 1 プロセッサ内で複数の TLB を扱わなければならず，また大きなキャッシュが必要になる．このように，従来の実スレッド管理モデルではプロセッサのワーキングセットを増大してしまうという問題がある.

また，スレッド管理機構を実現するという観点から見ると，Xeon 上で動く Linux では Linux Thread としてカーネルレベルスレッドが実現されている．しかし，前述したとおりスレッド制御を行うたびにカーネルモードに遷移しなければならず，オーバーヘッドが生じるという問題がある．Xeon プロセッサでの管理モデル上で 2 レベルスレッドを実現しようとすると，カーネルスレッドを複数ユーザレベルで管理することになる．このモデルでは並列実行も可能になり，ユーザレベルでスレッド制御を行うため，オーバーヘッドは軽減される．しかし，カーネルスレッドを生成，削除するためにはやはりカーネルヘドメイン切り替えを行わなければならない．これは，協調型スレッドモデルでも同じ問題がある.

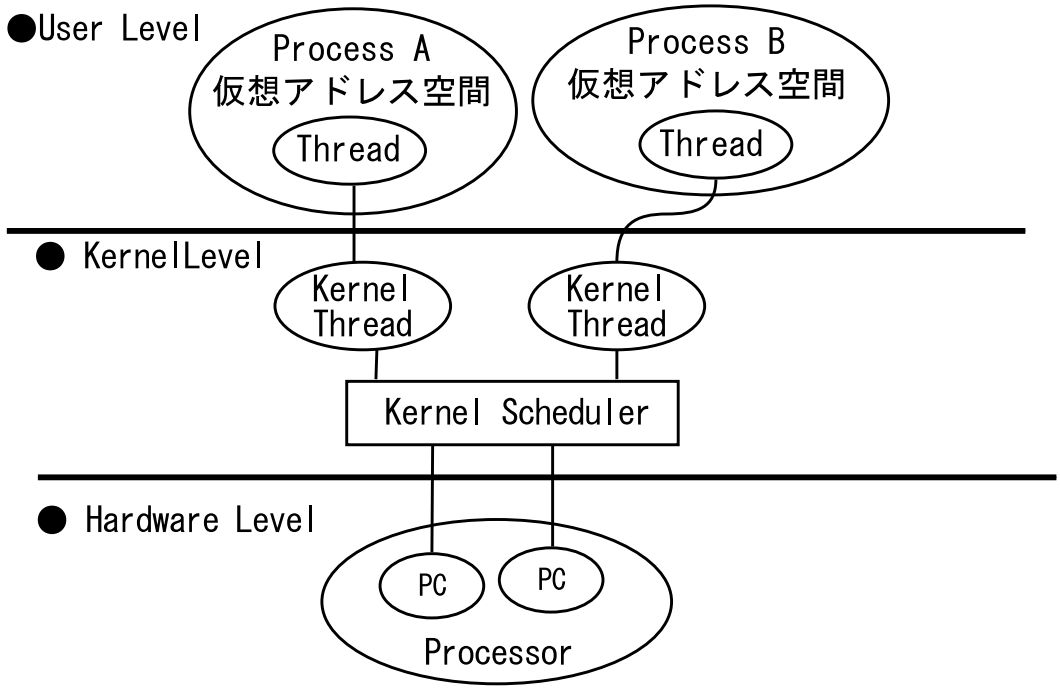


図 2.4: Xeon プロセッサの処理モデル

また，複数のプロセスが複数のカーネルスレッドを所持していた場合，たとえば図 2.3 のようなモデルを考えると，一つのアドレス空間に複数のカーネルスレッドが割り当てられる場合がある．このときは，できるだけ同一プロセスのカーネルスレッドがプロセッサを占有することが望ましい．こうすることでキャッシュのヒット率や TLB などを共有できるからである．このような工夫はカーネルスレッドスケジューラが行う必要がある．しかし，このような工夫を行っても他のアドレス空間のカーネルスレッドが一つの実スレッドを占有してしまうと，前述したワーキングセットが増大するという問題点は残る．

2.2.2 マルチスレッドアーキテクチャにおけるスレッドの排他制御・同期

SMT アーキテクチャは，複数の実スレッドを演算気を共有しながら並列に実行するため，実スレッド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，他の実スレッド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．

2.3 議論

本章で述べた問題点についてまとめた上で議論を行う．

スレッドモデルの検討

ユーザスレッドモデル，カーネルスレッドモデル，2 レベルスレッドモデル，協調型スレッドモデルがある．本研究ではシステム全体の性能向上を目指すため，協調型スレッドモデルを採用する．

実スレッドの管理モデル

システムソフトウェアが実スレッドをどのように管理するかについて，カーネルがすべてそれを管理するとオーバーヘッドが生じる．また，同一プロセッサ内で複数のアドレス空間が存在すると，ワーキングセットが広がり処理性能の問題となる．

マルチスレッドアーキテクチャプロセッサを効率よく利用するために，1 プロセッサ内の実スレッドはアドレス空間を共有することとする．また，実スレッド管理のオーバーヘッドを避けるため，ユーザレベルで実スレッドの管理を行うことができるようにする．

ユーザレベルで実スレッドの管理を行うとき，図 2.5 のように，スレッドを管理しないモデルが考えられる．これは並列化コンパイラが静的に実スレッドを制御するモデルである．本研究では，ユーザが明示的にスレッド生成を行い，実スレッドの管理をユーザレベルスレッドライブラリで行う，図 2.6 で示すモデルを採用する．

[*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．

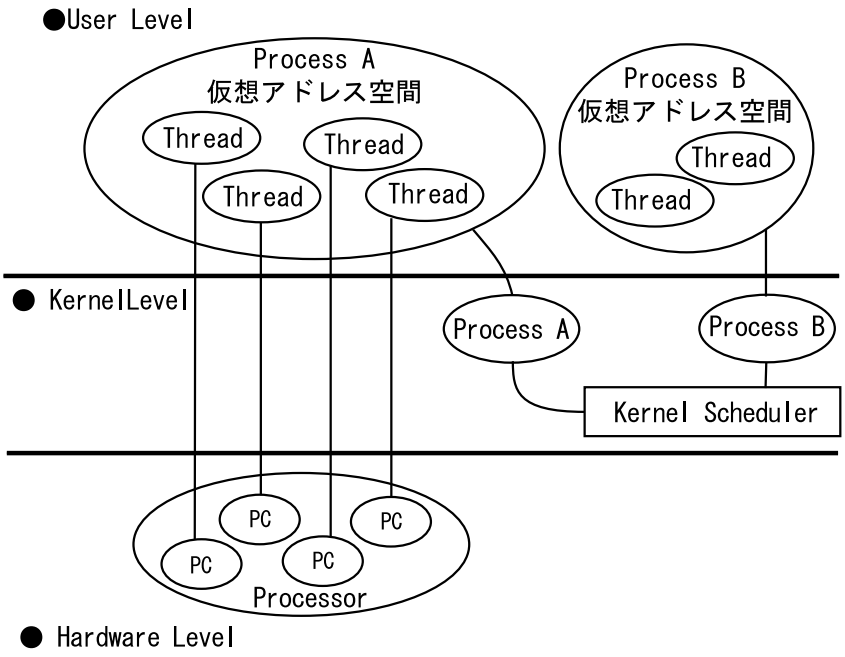


図 2.5: スレッドを管理しないモデル

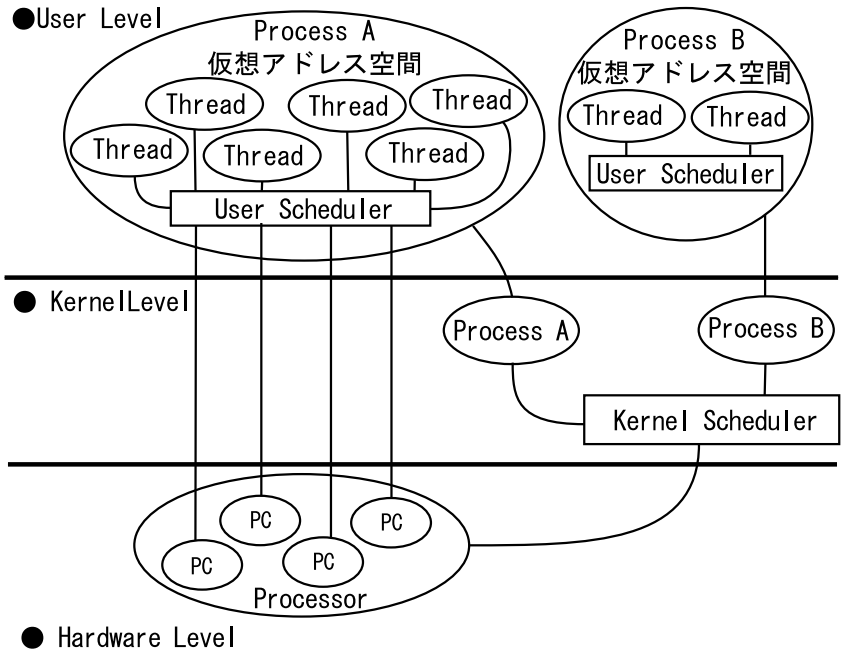


図 2.6: スレッドを管理するモデル

排他制御，同期機構の検討

マルチスレッドアーキテクチャでは従来の排他制御，同期機構のままでは性能向上を阻害する可能性がある．そのため，マルチスレッドアーキテクチャの特徴を生かした効率のよい排他制御，同期機構を検討する必要がある．具体的な検討とその方式については 5 章で論じる．

第 3 章

本研究の目標と設計方針

本章では研究の目的と方針について述べる．

3.1 本研究の目標

本研究では，並列実行可能なスレッドを実現可能とする，ユーザレベルで動作する効率のよいスレッドライブラリの作成を目標とする．

スレッドライブラリによってスレッドを生成，削除，排他制御，同期などの制御をアプリケーションプログラマが容易に利用できるようにする．また，これらの制御は軽量，高速に行うことを目標とする．

マルチスレッドアーキテクチャプロセッサが複数持つ実スレッドに，プログラマが生成したスレッドを割り当てることで，スレッドを並列実行させることができる．これにより性能向上を目指す．また，プロセッサの実スレッドの数よりも多くのスレッドを生成できるようにし，スレッドライブラリはこれを適切に管理する．

スレッド実行中に OS によるスレッドのブロッキングが発生しても，適切なスレッド切り替えが行えるように OS によるユーザレベルスレッドスケジューラの起動を行うための機構を実現する．これにより，スレッド実行の並列性を保ち，適切なスレッドスケジューリングを行うこと可能にする．すなわちスレッドの稼働率を高めることができる．また，同様の機構により効率のよいプリエンプティブなスレッド切り替えを提供する．

3.2 本研究の設計方針

本研究で提案するスレッドライブラリを次の方針で設計する．

- ユーザレベルでの実スレッド管理
  カーネルが実スレッドを管理すると，その管理コストは大きくなる．そのため，ユーザレベルで実スレッドを管理する．これにより柔軟で軽量な実スレッド管理が可能になる．また，ユー

ザレベルで管理することでスレッドの並列実行を可能にする．

- プロセッサの実スレッド制御機構を利用したスレッド制御
  実スレッド管理をユーザレベルで行うということは，プロセッサの実スレッド制御命令をユーザレベルで実行できるということを示す．このプロセッサの実スレッド制御命令を利用することで，スレッドライブラリが管理するスレッドの制御を高速に行う．
- 協調型スレッドモデルを適用する
  スレッド制御を高速に行うためにはユーザレベルでスレッド制御を行うことが求められる．ユーザレベルでスレッド制御を行う際，カーネル内での事象をユーザレベルは感知することができず，効率的なスケジューリングを行うことができないという問題点がある．そのため，OS と協調動作することで OS からの通知を受け取り最適なスケジューリングを可能にする．

第 4 章

システムの全体構成

前章で述べた本研究の方針に基づき，システムの全体構成がどのようになるかを示す．また，本研究を実現可能とするためのアーキテクチャを備えた OChiMuS PE プロセッサ，およびオペレーティングシステム Future について紹介する．

4.1 システム全体の構成

プロセッサアーキテクチャ，OS アーキテクチャ，ユーザレベルスレッドライブラリがどのように連携し，動作を行うかの全体構成を図 4.1 に示す．本節ではこの図が何をあらわしているかをスレッドライブラリとプロセッサ，OS との関係をもとに述べる．また，本章次節から，プロセッサとオペレーティングシステムについての解説を行う．

スレッドライブラリ

スレッドライブラリはアプリケーションソフトウェアからの POSIX Thread[5](Pthread) 仕様に準拠したスレッド制御関数呼び出しによって機能する．これは，スレッドの生成，削除，排他制御，同期などを行う．スレッドライブラリ内では，生成したスレッドの管理や利用していないスレッド管理ブロック (図中，ThMB で示すもの) の管理を行う．生成したスレッドはこのスレッド管理ブロックをそれぞれ一つ持ち，この領域にスレッドを管理するために必要な情報を格納する．スレッドライブラリはいつでも実行を再開することができる待ちスレッドの管理も行う．待ちスレッドとは，スレッドスケジューラが起動されたとき，復帰対象となるスレッドである．

スレッドライブラリとプロセッサ

スレッドライブラリはマルチスレッドアーキテクチャプロセッサである OChiMuS PE の実スレッドをユーザレベルで管理する．OChiMuS PE プロセッサについての詳細は後で述べる．実ス

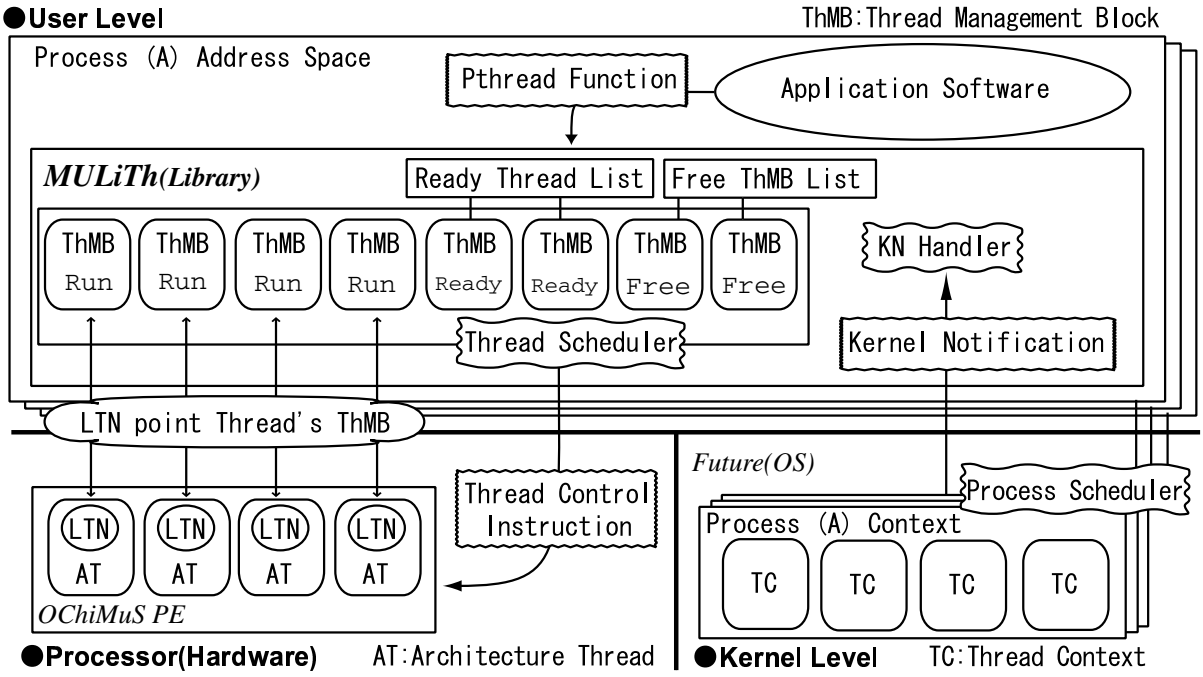


図 4.1: プロセッサ，OS，ライブラリ含めたシステムの全体像 (スレッド管理)

レッドの管理はプロセッサのスレッド制御命令を利用する．すなわち，ユーザレベルでスレッド制御命令が利用できるということである．このスレッド制御命令を利用することで，軽量なスレッド制御を可能にする．

スレッドライブラリは生成したスレッドを実スレッドに割り付けることでスレッドを並列実行を実現する．図中では 4 つのスレッドが Run，つまり実行中であることを示している．ThMB の先頭アドレスをプロセッサ実スレッドの特別なレジスタ (LTN：詳細は後述) に格納することで，スレッドの管理と実スレッドの管理を一元化する．つまり，プロセッサ内部にソフトウェアが管理するどのスレッドが実行しているかを情報として格納することができるということである．これにより，スレッドライブラリは実スレッドの管理を効率的に行うことができる．

スレッドライブラリとオペレーティングシステム

スレッドライブラリは OS Future と連携して協調型スレッドモデルを実現する．この協調機構を本研究では Kernel Notification という．スレッドライブラリは Kernel Notification Hander(KN Handler) を用意しており，OS はこの KNHandler へアップコールを行うことにより事象を通知して協調動作を実現する．

Future が管理するプロセスは OChiMuS PE プロセッサを仮想化したものと定義する．本研究で取り上げるプロセッサは同一プロセッサ内に一つのアドレス空間しか存在しないため，プロセス切

り替えが起こった場合は動作中の実スレッドをすべて退避し，アドレス空間を更新を行い，切り替え先プロセスで退避していた実スレッドコンテキストを復帰させるというものになる．このコンテキストを図中では TC という部分に格納している．ただし，Kernel Notification 機構では実スレッドの実行コンテキストの退避はユーザレベルの ThMB に対して行うため，この場合カーネルが管理する領域 TC へは次に実行する Kernel Notification Handler へ渡さなければならない情報を格納する．Kernel Notification 機構については本章および次章で議論する．

4.2 OChiMuS PE プロセッサ

この節では，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 [24] 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．

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．

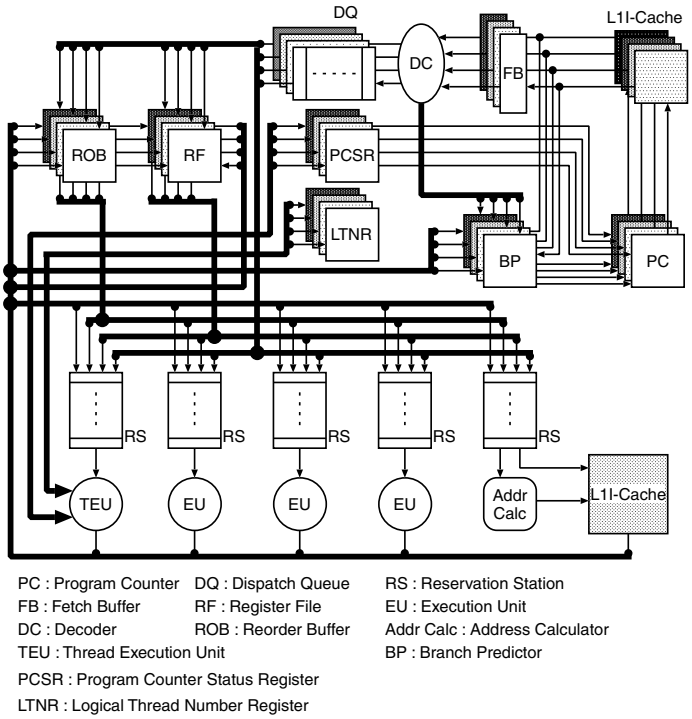


図 4.2: OChiMuS PE の構成 (中條研究室 河原氏提供)

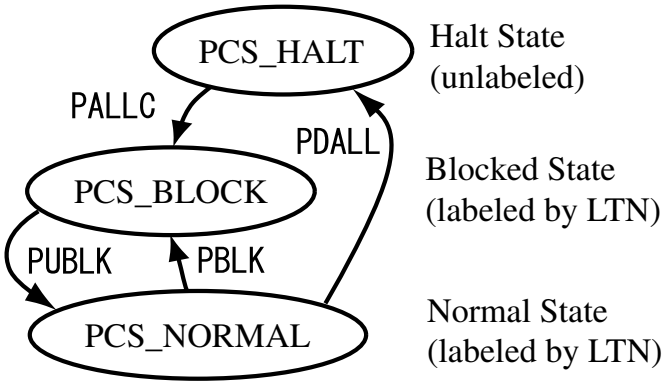


図 4.3: OChiMuS PE プロセッサの実スレッドの状態遷移

- 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．

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

4.2.2 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．

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．

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 Future OS

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

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

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[1] よりも効率的にこれを実現する．Kernel Notification の詳細は次章で述べる．

第 5 章

スレッドライブラリの方式設計

本章では, マルチスレッドアーキテクチャに適したスレッドライブラリである ***MULiTh***(Userlevel Library of Thread for Multithreaded Architecture) の設計について述べる.

5.1 スレッドライブラリの概要

スレッドライブラリ MULiTh は次の特徴がある.

(1) Pthread インターフェースである
スレッド制御のためのプログラミングインターフェースは，POSIX スレッド (Pthread)[5] 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.

(2) LTN を利用したスレッド管理
OChiMuS PE は実スレッドに LTN を設定することができる．これを利用したスレッド管理を行うことで，軽量なスレッド管理を可能にしている.

(3) マルチスレッドアーキテクチャプロセッサを利用したスレッド制御
実スレッドをユーザレベルで管理することができるため，スレッドはプロセッサに複数ある実スレッドに割り当て，並列実行することが可能である．また，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．これらのスレッドの制御は，OChiMuS PE のスレッド制御命令を利用することで，高速に行う.

(4) OS と協調するスレッドスケジューリング
OS と協調して動作することで，ユーザレベルでスレッド管理を行う際に発生する問題を回避する．具体的には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.

以降，本章ではこの特徴を実現するための設計について論じる.

表 5.1: 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

5.2 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[5][7](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 5.1 で示し，図 5.1 で示す宣言に基づいて利用する．この Pthread インターフェースについて，例を示しながら説明する．

5.2.1 スレッドの生成と削除，合流

スレッドの生成は pthread_create 関数によって行う．pthread_create には引数に (a) 生成したスレッドの識別子を格納するための変数へのポインタ (b) 生成するスレッドの属性を指定するためのパラメータ (c) 生成するスレッドの**スレッドルーチン**を示す関数へのポインタ，および (d) スレッド実行開始時にスレッドルーチンへ渡す引数が引数として渡される．スレッドが生成されるとスレッドルーチンの実行をはじめる．

スレッドの削除は pthread_exit 関数を呼び出すか，もしくはスレッドルーチンを終了すると行われる．スレッドルーチンでの返り値，または pthread_exit に対する引数を，このスレッドの返り値とし，そのスレッドは終了し，削除される．

スレッドの合流とは，あるスレッドが終了することを待つことをいう．これは pthread_join 関

```
/// 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 5.1: 作成した Pthread 関数のプロトタイプ宣言

数を呼び出すことで実行する．引数にどのスレッドを待つか，というスレッド識別子を指定し，終了を待つ対象のスレッドの返り値を得ることができる．

スレッドの生成, 削除, 合流を使用した例を図 5.2 で示す．このプログラムでは, main 関数内でスレッドを 10 個生成し，生成したスレッドの終了を pthread_join 関数によって待つ．スレッドルーチンへの引数に，何番目に生成されたか，という情報を与える．スレッドルーチン内では func という関数を呼ぶ．func 内では，そのスレッドが偶数番目に生成されたものであれば，pthread_exit

によってスレッドを終了し，static 変数 val のアドレスを返し，これ以後の動作を行わない．そうでなければスレッドルーチンを 0 を返り値として終了する．

5.2.2 スレッド間の排他制御

排他制御を行うためには pthread_mutex_lock と pthread_mutex_unlock を利用する．

排他制御は，たかだか一つのスレッドしか同時にある保護する必要のある領域（クリティカルセクション）を実行できないということを保証するものである．pthread_mutex_lock はロック変数を引数に実行すると，ロック変数が他のスレッドにロックを獲得されていなければすぐにロックを獲得して返る．あるいは，すでに他のスレッドがロックを獲得していれば，ロックが解放されるまで待ち，解放されたときに再度ロックの獲得を行い，最終的にはロックを獲得した状態で返る関数である．pthread_mutex_unlock はロックしていたロック変数を引数で渡し，そのロック変数のロックを解除するものである．

図 5.3 は排他制御を行うサンプルである．thread_func が複数のスレッドによって実行された場合，排他制御を行わないとグローバルカウンタの増加が正しく行われない可能性がある．そのため排他制御を行いグローバルカウンタのインクリメントを実行する部分をクリティカルセクションとして保護する．クリティカルセクションでは一つ以下のスレッドしか同時に実行しないことを保証する．

5.2.3 条件変数による同期

条件変数による同期を行うためには pthread_cond_wait と pthread_cond_signal を利用する．pthread_cond_wait は，引数として与えられた条件変数に，pthread_cond_signal による合図がくるまで待機する．一般的に，条件変数による同期は，ある条件が真ではないとき，スレッドを条件が変化するまで待機したいときに利用する．たとえば，複数スレッド間で共有するキューがあったとき，キューにデータがあれば（条件が真）その中から一つデータを取り出して処理を行い，データがなければ（条件が偽）条件変数により待機し，他のスレッドがキューにデータを格納したとき（条件を真にする）条件変数に合図を行い，待機していたスレッドを再開させる，などの場合に利用する．

条件変数をチェックするとき，mutex の保護下でこれを行う．これは，条件の判断を不可分に行うために必要となる．これを行わないと，条件を永遠に待ちつづけるスレッドが生成される可能性がある．pthread_cond_wait により待機を行うと，スレッドが保持していた mutex ロックは解除され，待機が解除されたときには mutex ロックは再獲得されるようになっている．

図 5.4 は条件変数による同期を行うプログラムを示す．あるスレッドが wait_func を実行する．このとき，global_condition で示される条件が偽であれば pthread_cond_wait によって合図を待つ．他のスレッドが signal_func を行えば，pthread_cond_signal によって待機していたス

```
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 5.2: スレッド生成，削除，合流を行うサンプルプログラム

```
pthread_mutex_t global_mutex_variable = PTHREAD_MUTEX_INITIALIZER;
int global_Counter;
void *thread_func(void *p){
  // enter critical section
  pthread_mutex_lock(&global_mutex_variable);
  global_Counter++;
  pthread_mutex_unlock(&&global_mutex_variable);
  // leave critical section
  return 0;
}
```

図 5.3: 排他制御を行うサンプルプログラム

レッドを再開することができる．また，条件の判定は mutex により不可分に行うことができる．

5.3 スレッドの管理

本節ではスレッド管理をどのように行うかについて示す．

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 5.5 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．

次に続く各項でスレッド管理方式について詳しく述べる．

5.3.1 スレッド管理ブロック

pthread_create 関数によって生成されるスレッドは，それぞれ管理するためのデータ領域をもっており，それをスレッド管理ブロック (ThMB:Thread Management Block) という．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;
```

```
pthread_mutex_t mutex;
pthread_cond_t  cond;
int global_condition = 0;
// 条件が真になるまで待つ
void wait_func(){
  pthread_mutex_lock(&mutex);
  while( global_condition == 0 ){
    pthread_cond_wait(&cond,&mutex);
  }
  // 条件が成立したときの処理を行う
  pthread_mutex_unlock(&mutex);
}

// 条件を真にして合図を行う
void signal_func(){
  pthread_mutex_lock(&mutex);
  global_condition = 1;
  // 条件が成立したことを知らせる
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

図 5.4: 条件変数による同期を行うサンプルプログラム

```
}ThreadManagementBlock;
```

ThreadContext は実行コンテキスト退避領域を示し，そのスレッドが中断するとき，情報を退避するために利用する．このデータには各種汎用レジスタと復帰後に設定するべきプログラムカウンタの値が含まれている．この領域はスレッドがプロセッサ上に存在しなくなるときに利用され，実際にプロセッサ上で実行しているときは利用されない領域である．

ThreadAttribute は，スレッド属性を示し，そのスレッドが現在どのような状況であるかを示すもので，スレッド操作一般で利用される．例えばそのスレッドがデタッチされた場合，そのスレッドがデタッチされたものであるという情報を記録する必要があるが，そのような情報を記録するために存在する．また，スレッド固有の情報，たとえば errno のような，従来のシングルスレッド環境では大域変数であったデータはこの部分に格納することでスレッドローカルなデータ領域として

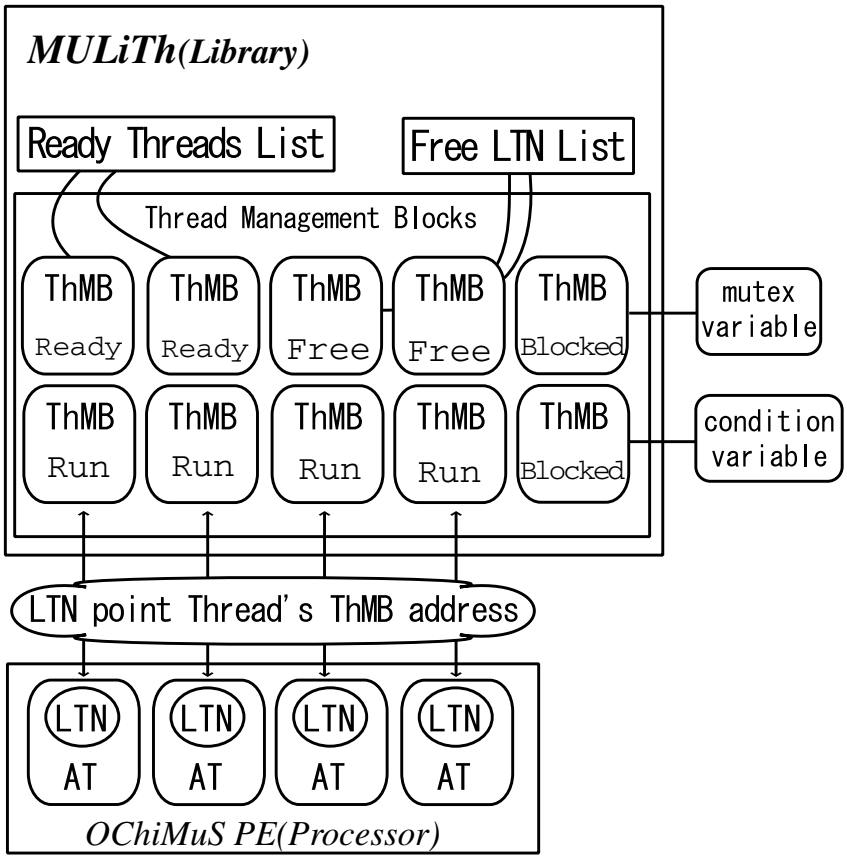


図 5.5: スレッド管理概観

管理することができる．

5.3.2 スレッド識別子と LTN

各スレッドは識別子を持ち，その識別子を利用することでスレッド操作の対象を決める．スレッドの識別子とは pthread_create 関数を実行したとき，作成したスレッドのスレッド識別子が得られる．また，pthread_self 関数により，現在実行しているスレッドの識別子を得ることができる．すべてのスレッドでそれが重複することはない．

このようなスレッドを一意に決めるためのスレッド識別子を，そのスレッドが持つスレッド管理ブロックの先頭アドレスとする．各スレッドはそれぞれ別のスレッド管理ブロックを持つため，その先頭アドレスもそれぞれ別のものになり，識別子として利用できる．

OChiMuS PE では，スレッド制御命令を実行するためには論理スレッドに特定するための論理スレッド番号 (LTN) が必要であるが，この LTN を MULiTh が管理するスレッド識別子を設定する．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．また，逆にプロセッサの LTN に設定した値がスレッド識別子であると定義できる．

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

図 5.6: pthread_self 関数の定義

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．図 5.6 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．[*1]．

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．

5.3.3 待ちスレッドリスト

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．

[*1] 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．

5.3.4 スレッドスケジューリング

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 [11] 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[*2]．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リングを行うことも考えられるので，スケジューリングによるオーバーヘッドとの兼ね合いをみて今後検討していきたい．

5.3.5 空き LTN リスト

MULiTh は利用していない ThMB を管理する必要がある．スレッド生成時には一つ利用していない ThMB が必要になり，スレッド削除時には，そのスレッドが利用していた ThMB が解放されたことを記録しなければならない．この管理はできるだけ高速に行うことが望ましい．そのため，MULiTh では，利用していない ThMB の先頭アドレスをリストとして管理している．MULiTh では ThMB の先頭アドレスは LTN であるため，このリストを空き LTN リストと呼ぶ．

MULiTh では現在，空き LTN リストをスタック構造で管理している．これは，利用していない ThMB に対して LIFO アクセスが可能になる．最後に利用していた ThMB を使うことによりキャッシュのヒット率向上が期待できるためである．また，スタック構造にすることにより，すべての ThMB の状態を検索する必要がなくなり，高速に空き LTN の管理ができる．

5.4 プロセッサ命令を利用するスレッド制御

MULiTh は OChiMuS PE のスレッド制御命令を利用して高速なスレッド制御処理を可能にしている．本節ではそれらの設計について示す．

5.4.1 スレッド生成

スレッド生成を生成する処理，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

[*2] 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．

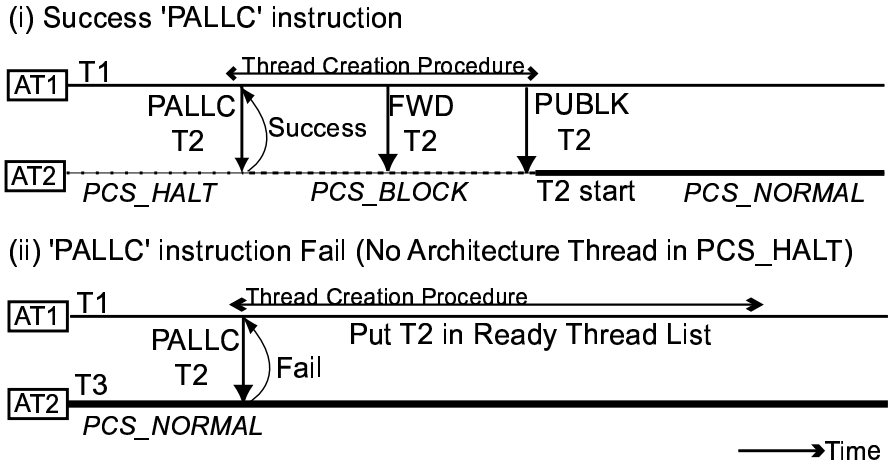


図 5.7: プロセッサ命令を利用したスレッドの生成

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，C 言語風の擬似コードで書くと次のようになる．

```
ltn = 空き LTN リストから一つとる
if( ltn が取得できなかった ){
  スレッド生成の限界として，エラーを返す
}
プロセッサ・スレッド生成命令の PALLC を実行する
if( PALLC 成功 ){
  // 実スレッドが一時停止状態で確保される
  プロセッサ・レジスタ転送命令である FWD により，初期値を設定
  プロセッサ・停止解除命令により生成されたスレッドが動作をはじめる
}
else{
  // PALLC 失敗：停止状態の実スレッドは存在しなかった
  ltn で示される ThMB に，実行開始時のスレッドコンテキストを格納する
  待ちスレッドリストに生成したスレッドを追加する
}
スレッド生成成功を返す
```

図 5.7 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 5.7 の (i) はプロセッサ・スレッド生成命令が成功したときを示している．PALLC 命令は，PCS_HALT 状態のスレッドがあれば，それを指定した LTN に割り当て，PCS_BLOCK 状態にする命令である．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

```
// thread entry point
static void thh_create_helper(
  int dmy1,int dmy2,
  void *(*start_routine)(void *),void *arg)
{
  // 関数呼び出し
  void *ret = start_routine(arg);
  pthread_exit(ret);

  asm("break"); // not reachable
}
```

図 5.8: スレッドのエントリーポイント

を対象に割り当てを行う．AT2 は，LTN を T2 に設定され，PCS_BLOCK 状態になる．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レジスタ値の転送を行う．設定する初期値はスタックポインタやスレッド実行に必要なパラメータなどである．その後 PUBLK 命令を発行し，スレッド T2 の実行を開始する．これにより，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 5.7 の (ii) は，プロセッサ・スレッド生成命令が失敗したときを図示している，PCS_HALT 状態の実スレッドが存在しなかったため，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 成功時にくらべ，コンテキストの保存など，メモリアクセスが増える．

なお，現状では生成されたスレッドへ設定するスタックは静的に確保した静的な領域である．

生成されたスレッドのエントリーポイント

生成されたスレッドは，図 5.8 に示す関数から開始する．

引数の最初二つは pthread_create 関数の引数にあわせるためのダミー変数である．start_routine，arg 引数は pthread_create で渡されたスレッドルーチン，およびその引数である．この関数はまず，引数として渡された start_routine 関数を，arg を引数として実行する．start_routine 関数が返るときはスレッドルーチンがリターンしたときである．このタイミングで pthread_exit を行う．これにより，スレッドルーチンがリターンしたときに pthread_exit の処理を行うことができる．

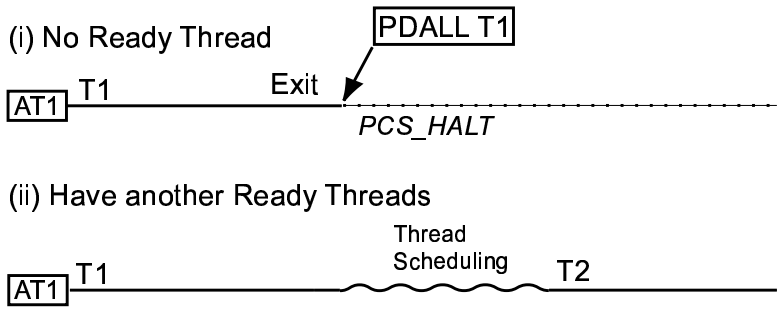


図 5.9: プロセッサ命令を利用したスレッドの削除

5.4.2 スレッドの削除

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[*3]．

削除処理は，次の順序で行う．

空き LTN リストに自スレッドの LTN を格納
スレッドスケジューリング
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．

図 5.9 ではスレッド削除の様子を示している．(i) では待ち状態のスレッドがないため PDALL 命令を実行している．(ii) では待ち状態のスレッド T2 へ T1 からスレッド切り替えがおきたことを示す．

5.4.3 排他制御，同期機構

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

[*3] 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．

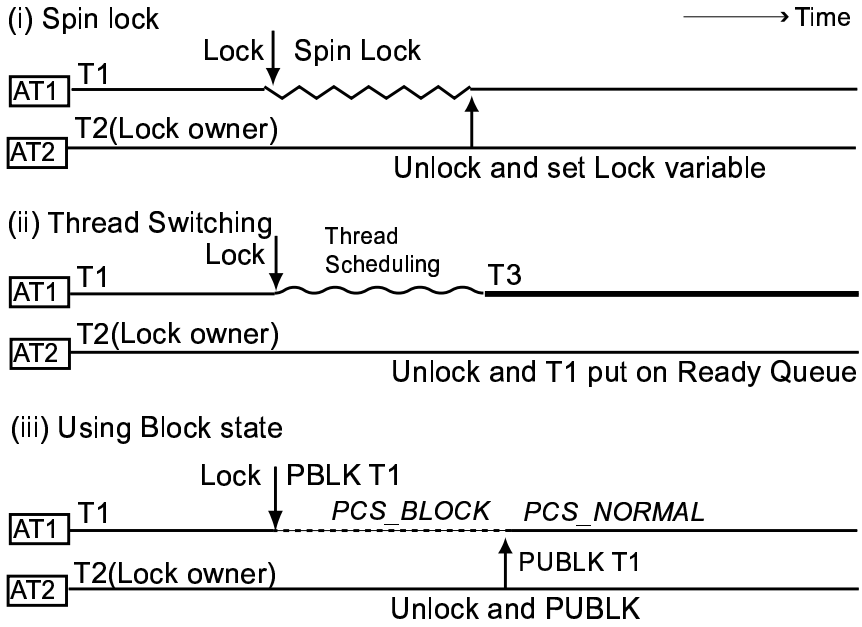


図 5.10: Mutex Lock の方式比較

(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

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，本方式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 5.10 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 5.10 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を行い，スレッド T3 へ切り替わったことを示している．(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，スレッド T1 がそのロックの解放を待っているとき，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．

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，一つの実スレッドを PCS_BLOCK とするため，すべての実スレッドをブロック状態にする危険性がある．また，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，待ち状態のスレッドが存在する場合は待ち状態のスレッドへ切り替えを行うようにした．[*4]

しかし，本手法でも PCS_BLOCK 状態のスレッドが実スレッドを占有してしまう，という問題は残る．たとえば，ある実スレッドを PCS_BLOCK にしたあと，スレッド生成などで待ち状態のスレッドができた場合，このブロック状態には生成したスレッドを割り当てることができない[*5]．

この問題点を解決するにはいくつかの方法があるが，ユーザレベルだけ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，この本問題を 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で述べる．

排他制御

pthread_mutex_lock は次の手順で行われる．

```
ロック獲得を試みる
if(ロック獲得成功){
  ロックを獲得し，かえる
}
else{
  // すでに他のスレッドにロックが獲得されていた
  ロック変数に記録されているロック待ちスレッドリストに自分を登録する

  if(待ち状態のスレッドがある){
    待ち状態のスレッドへスレッド切り替え
  }
  else{
    PBLK 命令によってスレッドを一時停止
    // 他スレッドによってロックを解除されたときにはロックが獲得されている
  }
```

[*4] スケジューリングポリシによって待ちスレッドがあってもブロック状態へ遷移するように選択できるようにしてもよい．これについては現在検討中である．

[*5] PALLC 命令が対象とするのは PCS_HALT の実スレッドだけである．

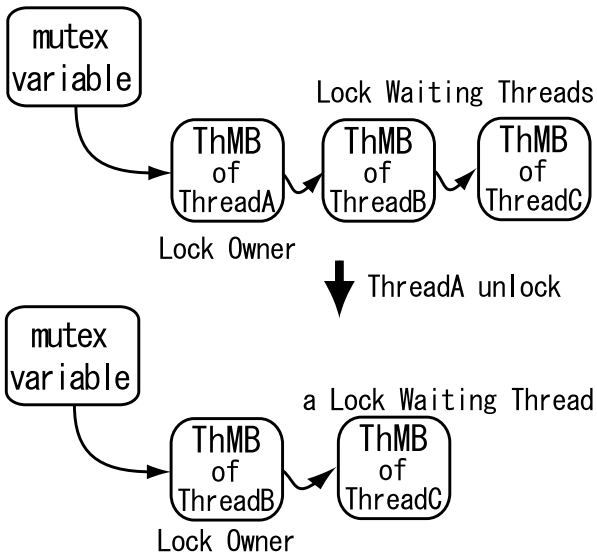


図 5.11: ロック変数に対して待っているスレッドのリスト

```
}
```

ロック変数に記録しているスレッドのリストとは図 5.11 のようになっている．図の上部は，スレッド A がロックを獲得しており，スレッド C が最後にリストへ加えられたスレッドであることを示している．

pthread_mutex_unlock によるロックの解除は次のような手順で行われる．

```
if(ロック変数にこのロックの解除を待つスレッドが登録されているか？){
  // されている
  ロック変数に記録されている待ちスレッドリストの先頭スレッド T をロックの所有者にする
  if(スレッド T に対し PUBLK 命令をかける){
    // 成功
    // 何もしない
  }
  else{
    // 失敗
    待ちスレッドリストにスレッド T を格納する
  }
}
else{
  // どのスレッドも解放を待っていない
  ロック変数を解放する
```

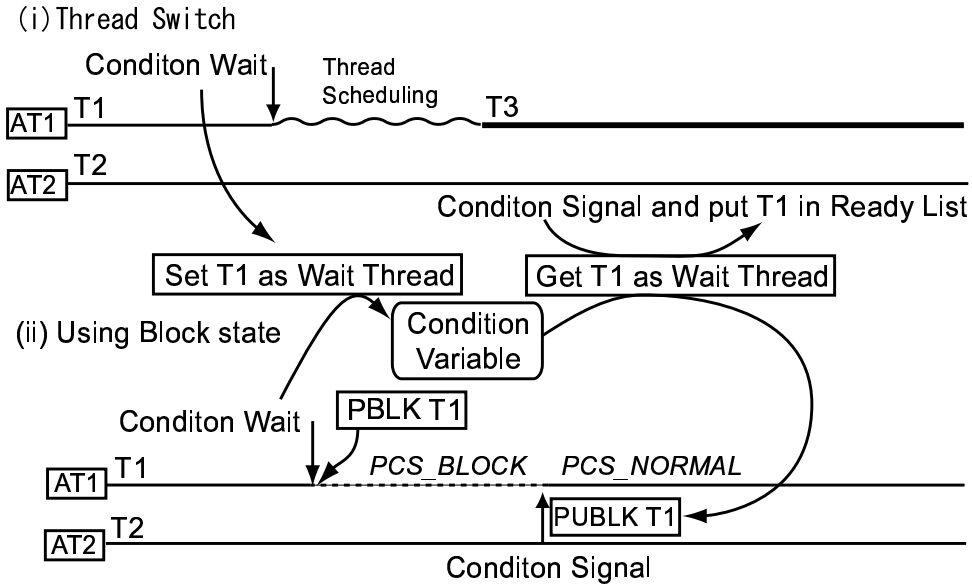


図 5.12: 条件変数による同期

```
}
```

図 5.11 の下部は，スレッド A がロックを解放し，スレッド B へロックの所有権が移ったことを示している．このアルゴリズムにより，mutex による排他制御が可能になる．

条件変数による同期

pthread_cond_wait，pthread_cond_signal による同期は mutex によるロックとほぼ同様のアルゴリズムである．mutex での lock では，ロックが誰にも所有されていなければロックを取得し，処理を続けることができるが， pthread_cond_wait では pthread_cond_signal によって合図されない限り，実行が再開されないという点で違いがある．

図 5.12 では，従来の条件変数による同期と本提案の同期機構について示しており，(i) では従来のスレッド切り替えによる同期，(ii) ではプロセッサブロック状態を利用した同期を行っている．(ii) では PUBLK 命令が実行されたときにすぐにスレッド実行を再開していることがわかる．

合流

pthread_join によるスレッドの合流は，pthread_cond_wait によるスレッドの待機とほぼ同様である．待っている対象のスレッドが終了したとき，pthread_cond_signal によって起こされるの

と同様にスレッドの待機が解除される．

5.4.4 並列実行のための排他制御

スレッド制御関数はすべて並列に実行できるようにしているが，なかには並列実行できないクリティカルセクションがある．たとえば待ちスレッドリストへのアクセスは同時にたかだか一つのスレッドしか行うことができない．これを実現するため，MIPS プロセッサアーキテクチャの test and set 命令を利用する．具体的には LL，SC 命令を図 5.13 のように定義したマクロで保護すべき領域（クリティカルセクション）を囲むことで実現する．

スレッド制御関数の中では，pthread_mutex_lock のような関数は利用することができない．とくに，pthread_mutex_lock 処理内も排他制御しなければならない部分があるため，排他制御の一番低レベルな test and set 命令を利用する必要がある．test and set 命令で利用するロック変数は，できる限りロックしている間隔が短くなるよう，それぞれのロックの契機に応じて用意する．

test and set による排他制御はスピンロックによるアルゴリズムであるため，マルチスレッドアーキテクチャでは他のスレッドの実行を阻害する可能性がある．そのため，OChiMuS PE プロセッサでは SLEEP 命令という，実スレッドの実行を一時停止する命令を用意するかどうか検討中である．SLEEP 命令を利用した test and set 命令でのスピンロックは図 5.14 のようになる．

プロセッサは一回のスピンロックのイテレーション中で LLSCLOCK_SLEEPCYCLE で示されるクロック数，実スレッドの実行を停止する．このように SLEEP 命令を挟むことにより，他のスレッドへの影響を抑えることができるのではないかと期待できる．この SLEEP 命令を併用した test and set 命令による排他制御についての性能評価は本論文の範疇外であるため行わない．

5.5 OS との協調

ユーザレベルにあるスレッドライブラリだけでは解決できない問題を，OS と協調動作することで解決した．本節ではこの方法について論ずる．

MULiTh は，OS 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．

(1) 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
(2) 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
(3) ブロックしていたスレッドが再開した場合
(4) 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

```
// lock
#define TH_LOCK(lock_addr) \
{ \
  asm volatile( \
                ".set noreorder\n"\
                "1:;"           \
                "ll   $2,(%0);" \
                "bnez $2,1b;"   \
                "li   $8,1;"    \
                "sc   $8,(%0);" \
                                \
                "beqz $8,1b;"   \
                "nop;"          \
                "2:;"           \
                ".set reorder\n"\
                                \
                : /* output */                    \
                : "r" (lock_addr) /* input  */    \
                : "$2","$8" /* broken register */ \
                );}

// unlock
#define TH_UNLOCK(lock_addr) {\
  *lock_addr = 0;\
}
```

図 5.13: test and set 命令による排他制御

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 [1] 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 [26] 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

```
#define TH_LOCK(lock_addr) \
{ \
  asm volatile( \
               ".set noreorder\n"\
               "j 2f;"          \
               "li   $8," LLSCLOCK_SLEEPCYCLE ";"  \
               "1:;"            \
               "sleep $8;"      \
               "2:;"            \
               "ll   $2,0(%0);"\
               "bnez $2,1b;"  \
               "nop;"           \
               "sc   $8,0(%0);"\
                                \
               "beqz $8,1b;"  \
               "li   $8," LLSCLOCK_SLEEPCYCLE ";"  \
               ".set reorder\n"\
                                \
               : /* output */                    \
               : "r" (lock_addr) /* input  */    \
               : "$2","$8" /* broken register */ \
               );}
```

図 5.14: SLEEP 命令を加えた test and set 命令による排他制御

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．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

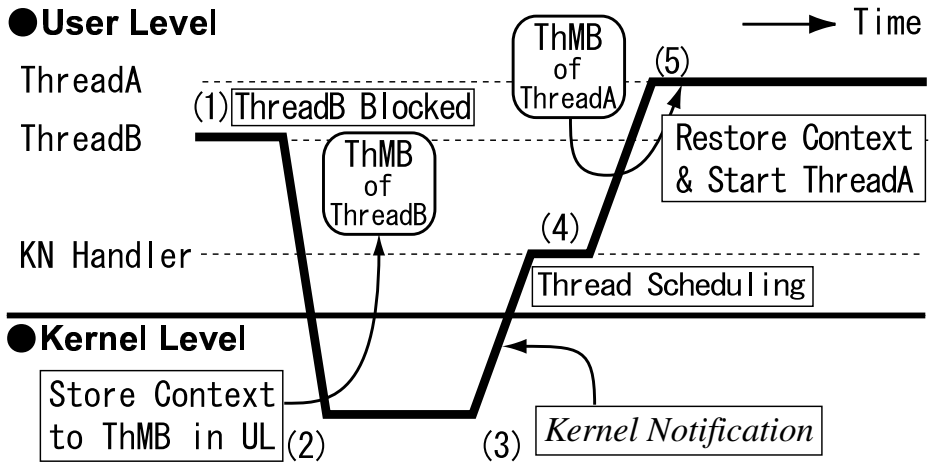


図 5.15: Kernel Notification の処理の流れ

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 からの通知を受け取ることができる．また，Scheduler Activation では事象の通知を行うために新たにカーネルスレッドを作成する．また，猪原らの提案でも Notifier カーネルスレッドを生成する．そのため，通知に本方式以上のオーバーヘッドがかかると考えられる．C-area によるユーザレベル，カーネルレベルの情報伝達は，ユーザレベルスケジューラがスレッドライブラリ内で管理するスレッドコンテキストデータ領域（MULiTh の ThMB に相当する部分）と C-area の管理を行わなければいけないため，スケジューラの構造が複雑になる可能性がある．本方式ではカーネルが直接 ThMB に対してスレッドコンテキストを退避するため，ユーザスケジューラは ThMB のみ管理すればよく，そのような複雑なユーザスケジューラを必要としない．

図 5.15 で Kernel Notification の処理を示す．スレッド B を実行中に例外などの事象が起こり，スレッド B の実行が中断される．スレッド B の実行コンテキストはスレッド B の ThMB へカーネルが退避する．スレッド B の ThMB のアドレスは実スレッドの LTN を取得すれば容易に得ることができる．LTN の取得はレジスタアクセスで済むため高速に行うことができる．カーネル内での処理が終了すると，その実スレッドはユーザレベルへドメインを戻し，KNH をアップコールにより実行している．KNH ではスレッド B を待ちスレッドリストへ加え，スケジューリングを行いスレッド A を復帰する．

この機構を用いるためには ThMB のスレッドコンテキスト退避領域の構造についてスレッドライブラリと OS が共通の仕様をもたなければならないため，移植性は犠牲になる可能性がある．しかし，データ構造として共通の認識をもたなければならない部分はどのレジスタが何番目に格納されるか，という点だけであるのでこれが機能的な問題になることはないと考えられる．

ユーザレベルのみでこれらのことを対処しようとすると，ブロックするようなシステムコールにはラッパ関数を用意してブロックするようなシステムコールを利用せず，代替手段を用いるなどの工夫が考えられる．しかし，そのようなスレッドライブラリはプログラミングが非常に複雑になる

可能性がある．また，ページフォールトなどが発生したなどの事象の通知を受けることができないため，それらを考慮したスケジューリングは不可能である．Kernel Notification の機構により，システムコールはシングルスレッドモデルと同様に実行することができる．

5.5.1 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 KNH が発行されることが保証される[*6]．

本機構を実行するコストは，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だけであるため，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，効率よく実行することができる．[*7]

図 5.16 では，PTL[21] 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 回ずつしか行われない．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．

5.5.2 ブロック，停止している実スレッドの扱い

前項で述べたように，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．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，ブロック状態のスレッドの退避が実現できる．

[*6] 後述するクリティカルセクションでの競合回避中の状態を除く．

[*7] プロセス切り替えが発生しない，プロセスが OS 内に一つしかないような状態では，OS はどのような挙動をとるべきかは現在検討中である．

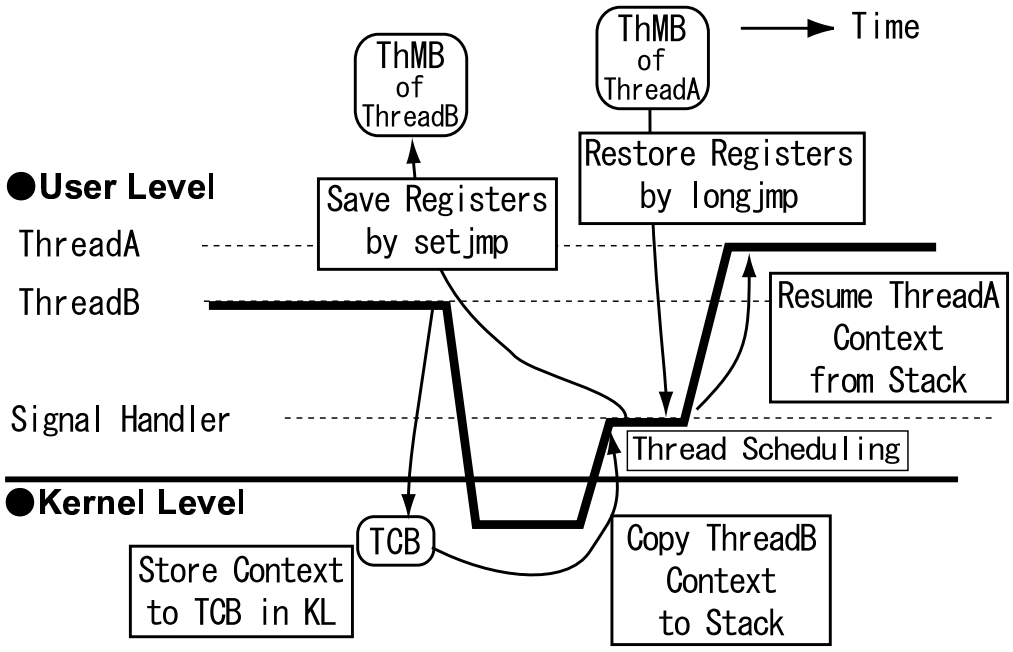


図 5.16: 従来のシグナルの利用によるプリエンプティブなスレッド切り替え

図 5.17 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（ロックを解除するなど）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．

5.5.3 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．また，KNH でスケジューリングを行うときにアクセスする待ちスレッドリストにアクセスする際，このデータへのアクセスをシリアライズするために MIPS の test and set 命令を利用してスピンロックによる排他制御を行うが，このときに利用する変数がロック状態のままそのスレッ

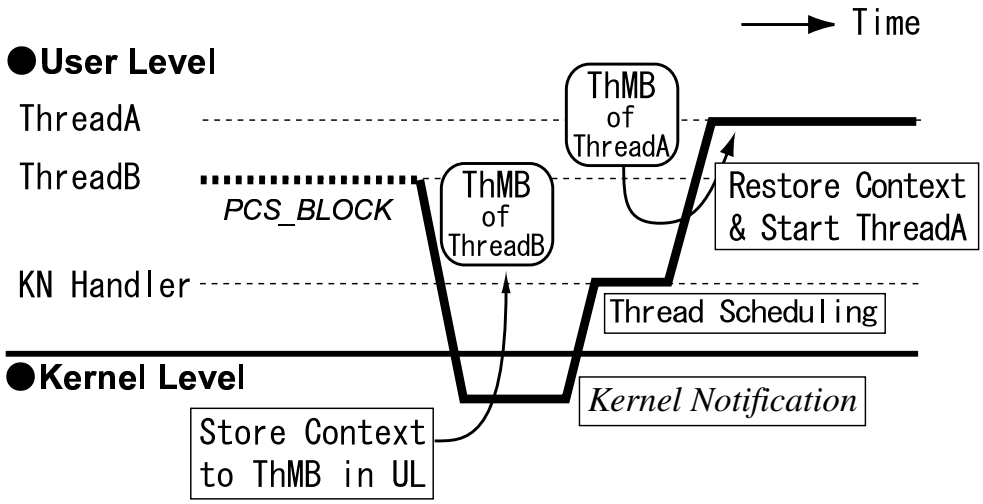


図 5.17: ブロック状態のスレッドの退避

ドがカーネルによって退避すると，KNH でスレッドスケジューリングを行うとき，そのロックを獲得することができず，デッドロックを起こしてしまう．このようなことが起こらないよう，ThMB へのアクセスなど，カーネルによる非同期な Kernel Notification を行われては困る場合には競合を回避する工夫が必要がある．

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．つまり，スレッド識別子 0 を特別な値とすることでカーネルにこのスレッドを Kernel Notification 機構を実行してはいけない，ということを伝える．

スレッドライブラリがそのようなクリティカルセクションに入るとき，実行する実スレッドの 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 へアップコールを行わず，カーネル内に退避したコンテキストで復帰する．その様子を図 5.18 に示す．図中では，スレッド B がクリティカルセクション実行中，割り込み例外などが発生してカーネルへドメイン切り替えが起こった様子を示している．カーネル内に用意したタスクコントロールブロック (TCB) に実行コンテキストを退避し，カーネル内での操作を行う．ユーザレベルへ復帰時は TCB から実行コンテキストを復帰し，スレッド B の実行を再開する．このとき，KNH は実行しない．この場合のカーネルによる実行コンテキストの退避は従来の OS と同様の方式である．

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．競合を回避する場合，KNH へ飛ぶことはないため，このようなクリティカルセクション中ではカーネル内でブロックするようなシステムコールの利用をしないことはもちろん，このクリティカルセクションの長さをできる限り短くしなければならない．

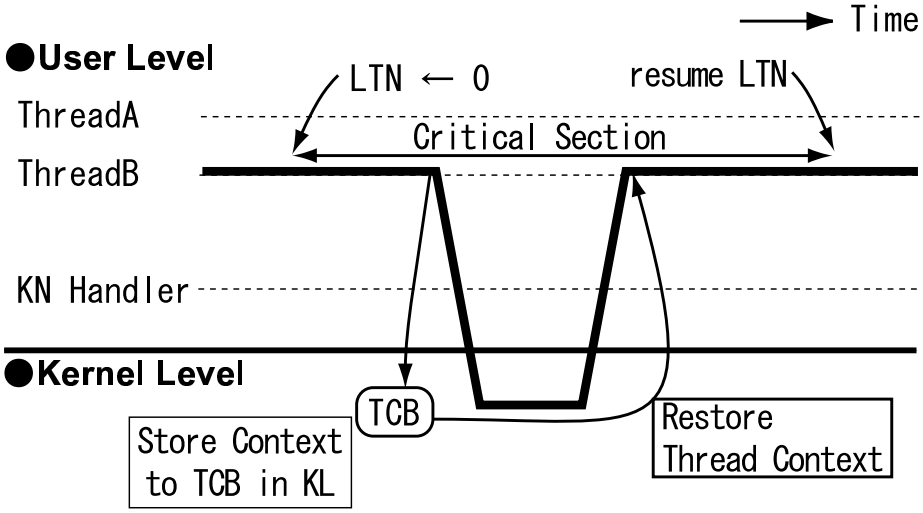


図 5.18: カーネルとスレッドライブラリとの競合の回避

第 6 章

評価

本章では作成したスレッドライブラリの性能評価について示す.

6.1 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.

6.2 評価環境

評価は本学中條研究室で作成している実行駆動型シミュレータ **MUTHASI**(MUltiTHreaded Architecture SImulator)[24] を用いた．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させずに評価を行った.

シミュレータの設定は表 6.1 とした.

6.3 並列実行による高速化の評価

本節では，マルチスレッドアーキテクチャプロセッサによる並列実行をスレッドライブラリで実現し評価した，その結果について述べる.

表 6.1: シミュレータの設定

AT Number	1	2	4	8
size of Fetch Buffer	32	16	8	4
Fetch Instructions	16	8	4	2
Simple ALUs	2			
Complex ALUs	1(Mult : 12 cycle delay / Div : 32 cycle delay)			
Cache	None			

6.3.1 画像縮小プログラム

図 6.1 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．対象とする図形をいくつかの領域にわけ，それらの領域の画像縮小を並列に行った．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．

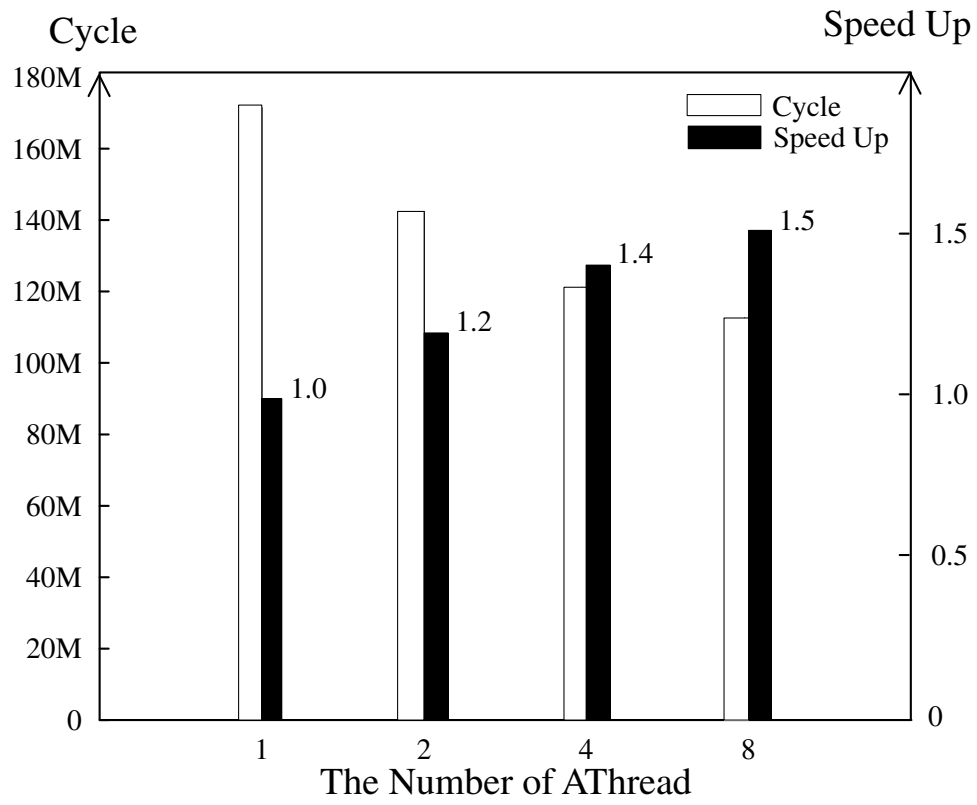
6.3.2 行列演算プログラム

図 6.2 は，2 つの行列の積を求めるプログラムを前項と同様に並列実行したものである．掛け算対象の行列をいくつかの行列にわけ，それらを並列に実行した．

並列化によって速度向上がないという結果となった．これは，行列の積は掛け算が主であり，プロセッサの掛け算器が一つしかないためであると考えられる．つまり，このような演算器の競合が発生するような同一の計算を繰り返すような処理をマルチスレッド化しても，マルチスレッドアーキテクチャでは速度向上を望めない場合があるということがいえる．

6.4 スレッド制御の評価

表 6.2 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．



グラフ内の数字は 1AT と比較した速度増加率

図 6.1: 画像縮小プログラムを並列実行した結果

スレッドの生成

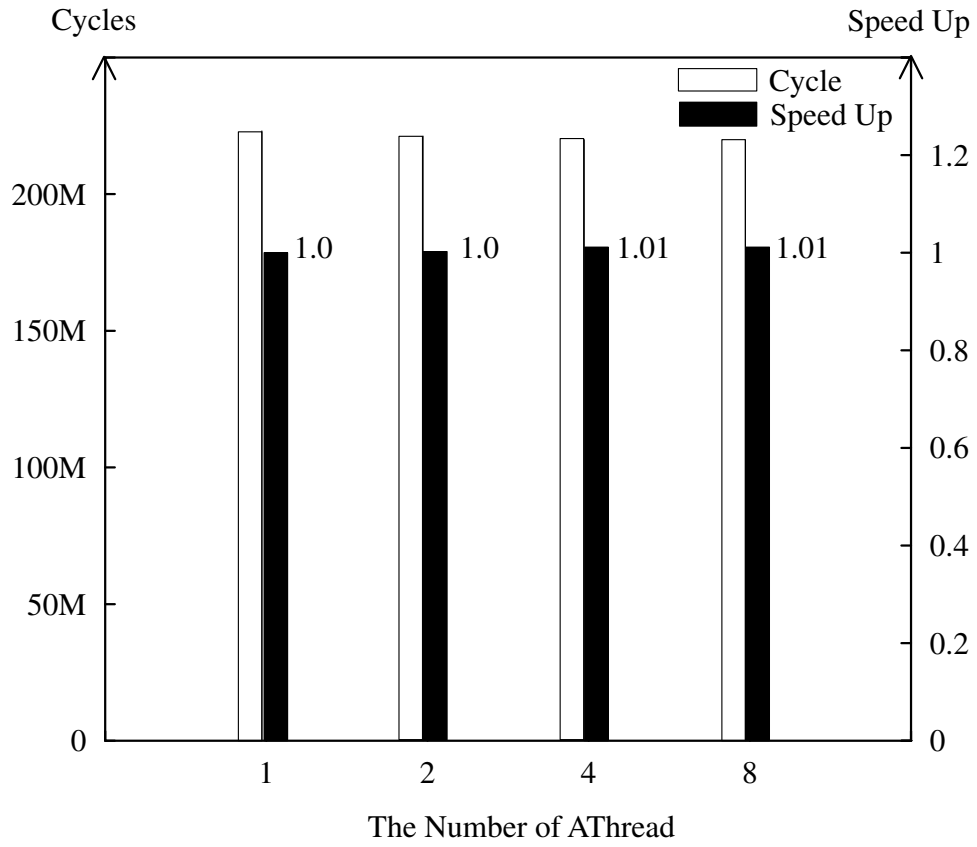
(1) は PALLC 命令が成功した場合，(2) は従来のスレッドライブラリの動作，または PALLC 命令が失敗したときの動作である．(2) では実際に生成したスレッドが動作するには，加えてスレッド切り替えのオーバーヘッドがかかる．

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だけですむため，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．



グラフ内の数字は 1AT と比較した速度増加率

図 6.2: 行列の積を求めるプログラムを並列実行した結果

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．

評価した処理を図 6.3 に示す．4 つのスレッドを生成し，それぞれのスレッドでグローバル変

表 6.2: 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)

数に格納されている変数を増加させるプログラムを作成し，測定した．増加処理中はスレッドを pthread_mutex_lock により，排他制御している．排他制御にして保護する時間を，短時間 (増加を 10 回繰り返す)，長時間 (1000 回繰り返す) と，2 種類の場合を評価した（図中の LOCK_TIME が時間を表す)．性能の数値はこのプログラムの実行時間である．

短時間の処理では，従来のスピンロック方式とプロセッサのブロック状態を利用した方式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，スピンロック方式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ると考えることができる．具体的には長時間ロックを保持していると 1.1 倍スピンロック方式に比べ性能が向上した．

本評価では，時間という単位を変数のカウントを増加させる 1 回の繰り返しとしているため，長時間といっても細粒度〜中流度の処理である．この粒度が大きくなると，スピンロックの場合はますます性能劣化が起こると考えられ，提案方式の有用性がさらに増すと期待できる．

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

```
#define LOCK_TIME 10 or 1000
volatile int global_count = 0;
void *thread_func(void *p){
  th_mutex_lock(&global_MutexLock);
  for(i=0;i<LOCK_TIME;i++){
    global_count++;
  }
  th_mutex_unlock(&global_MutexLock);
}

int main(){
  int i;
  for(i=0;i<4;i++){
    pthread_t th;
    pthread_create(&th,0,thread_func,0);
  }
  return 0;
}
```

図 6.3: 排他制御の性能評価用プログラム

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，提案方式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．

今回の評価では，シグナル処理についてはコンテキストのコピーのみをシミュレーションしたもので，シグナルマスクの検査やシグナルハンドラの検索などの処理は行っていないため，実際のシステムではシグナルでの方式はより性能が落ちると考えられる．

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[26] による通知のコンテキストコピー部分のみを実装し，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．

Scheduler Activations は，本来ならば事象通知のためのアップコールを行うとき，カーネルスレッドを新たに生成する．そのオーバーヘッドが含まれていないため，通常のシステムではこれより遅いと考えられる．

6.5 考察

本章では提案しているスレッドライブラリ MULiTh を利用したアプリケーションソフトウェアの並列実行による処理性能の向上についての評価と，スレッド制御についてのオーバーヘッドについて評価を行った．

アプリケーションソフトウェアの MULiTh を利用したマルチスレッド化は，Pthread を利用したプログラムで作成したため，ソースコードレベルでの互換性を維持している．また，実際に他の Pthread をサポートしている処理系で実行し，プログラムの検証を行うことができた．このように，Pthread というライブラリを通じてマルチスレッドアーキテクチャプロセッサの機能を容易に利用することを実現できたことを確認した．

性能面では，画像縮小のような加算などの単純な演算が大半を占めるような例では，並列実行によって演算器利用率を向上させることができ，全体の性能が向上した．しかし，行列の掛け算のような複雑な演算（掛け算）が多い計算では，掛け算を行う Complex ALU に対して競合が起こり，性能向上が望めないことがわかった．このような，一つの演算器に対してだけ競合が発生し競合が起こるような計算では演算器の有効利用が望めず SMT アーキテクチャでは性能向上が認めらないことを確認した．並列実行するスレッドがどの演算器を利用するかを前もって予測することは難しい．そのため，同一の処理を複数並列実行させるような，今回の評価における行列の掛け算の例のような計算モデルは控えたほうがよいことがわかった．SMT アーキテクチャでは，パイプラインパラダイムのような，複数の別々の処理を行うスレッドが並列実行するようなモデルで性能向上が期待できる．

スレッドライブラリの性能については，プロセッサのスレッド制御命令が利用できる場合には明らかに効率がよくなることがわかる．これを考えると，実スレッド数のスレッドを生成し，同期機構などを利用してパイプライン処理，ワークパイルのような処理などを行うと性能向上が期待できる．

OS との協調機構は，従来のものにくらべても十分高速であることがわかった．このため，効率的なスレッドスケジューリングができることが確認できた．

今回の評価では行わなかった，スレッド制御命令を多く利用し，および OS との協調機構により実行するアプリケーションソフトウェア，たとえばコネクションを多数開くため，多くのスレッドを生成するようなサーバ系ソフトウェアの評価を行うことが今後の課題である．

第 7 章

結言

本章では本研究の成果と適用可能性について述べ，今後の課題についてまとめる．

7.1 本研究の成果

本稿では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について検討し，実際に実現した．そして，それを評価した結果を示した．

評価の結果，スレッドを生成することで，プロセッサの実スレッドを並列実行させ，性能を向上させることができることがわかった．また，プロセッサのスレッド制御命令を利用し，軽量なスレッド制御が行えることが確認できた．具体的には，次のような成果が達成できた．

(1) OChiMuS PE の実スレッドをユーザレベルで管理することで，スレッドの並列実行が実現でき，マルチスレッドプログラムの並列実行による性能向上を確認した．画像縮小プログラムでは最大 1.6 倍の性能向上を確認した．
(2) OChiMuS PE のスレッド制御命令を利用することで，軽量なスレッド制御が実現できた．たとえばスレッド生成のオーバーヘッドは従来の方式にくらべ最大 4 割削減することができた．
(3) 実スレッドのブロック状態を利用することで，他の実スレッドの実行を阻害することなく効率的な排他制御，同期機構を実現できた．排他制御で 1.2 倍，同期機構では 4 倍の性能向上を図ることができた．
(4) OS と協調する機構 (Kernel Notification) によりカーネル内でのブロックの回避などを利用することができた．また，その連携についても OChiMuS PE の LTN を利用することでユーザレベル，カーネルレベルの効率的なな情報の伝達を達成できた．従来の方式に比べ少なくとも 1.4 倍の効率で伝達ができることを確認した．
(5) OS との協調機構を利用するスレッドのプリエンプションにより，シグナルによるスレッド切り替えに比べ最低でも 1.6 倍の性能向上を図ることを確認した．
(6) スレッド操作を Pthread インターフェースにより利用可能にすることができた．

7.2 本研究の適用可能性

本研究は OChiMuS PE プロセッサを対象として議論を行ったが，本論文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．

7.3 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．

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．

本研究ではマルチスレッドアーキテクチャにおける Pthread ライブラリの構築を目的としたが，Pthread インターフェース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生かすことができなかった部分もある．たとえばスレッドの生成はプロセッサアーキテクチャ上での実スレッド割り当ては 1 命令で実現できるが，これを利用するには アプリケーションプログラマは pthread_create 関数を呼び出さなければならない．しかし，pthread_create 関数は属性値の指定やスレッド管理ブロックの初期化など，その他の処理のオーバーヘッドが大きいため，細粒度スレッドを利用するには不向きである．そのため，他のインターフェース，たとえばすでに初期化したスレッド管理ブロックを用意しておき，実スレッド割り当てに成功すればそれを利用し，失敗すればエラーを返して終了するような Pthread 非互換のインターフェースを用意するなどの工夫が考えられる．このように，Pthread の枠にとらわれないスレッドライブラリの開発が今後の課題である．

また，並列化の手法には他にも OpenMP のような処理系，また並列化言語による細粒度スレッドを可能にする言語処理系によるもの，並列化コンパイラによるものなど，さまざまなものがある．このような手法にマルチスレッドアーキテクチャプロセッサを利用し，処理性能を目指すこと，とくに言語処理系での方式検討は興味深いテーマである．

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．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 年の佐藤 恵巳氏には筆者の苦手な英語について相談にのっていただきました．深く感謝いたします．

同志田 隆弘氏，下屋敷 太一氏には筆者の知らない知見を広げるきっかけを作っていただきました．深く感謝いたします．

本学並木研究室博士前期課程 2 年の笠井 秀一氏，駿藤浩之氏，堀口努氏，同博士前期課程 1 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．深く感謝いたします．

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．

本学中條研究室学部 4 年の加藤 義人氏，大和 仁典氏には，研究について相談に乗っていただき，とくにハードウェアについての造詣を深くすることができました．深く感謝いたします．

同中條研究室博士前期課程 2 年の河原 章二氏には，辛抱強く本研究の対象となるプロセッサアーキテクチャについてご指導いただき，また有益な助言を多くいただきました．深く感謝いたします．

本学並木研究室博士後期課程 1 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，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，また原稿のチェックなどをしていただきました．深く感謝いたします．

並木 美太郎助教授には，研究室配属以前からご指導いただき，また配属後も日常の細かい点から研究についての大きな視点まで，実にたくさん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や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．心より感謝いたします．

最後に，不規則な生活の筆者を支えてくれた家族に感謝いたします．

参考文献

[1] 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.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] E. Mohr, D. Kranz, and Jr. Halstead. LAZY TASK CREATION: A TECHNIQUE FOR INCREASING THE GRANULARITY OF PARALLEL PROGRAMS. Technical Report MIT/LCS/TM-449, 1991.

[10] 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.

[11] 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.

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

[13] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation, 1994.

[14] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/MP: Integrating futures into calling standards. In *Principles Practice of Parallel Programming*, pp. 60–71, 1999.

[15] 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.

[16] 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.

[17] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPoPP*, pp. 208–217, San Diego, CA, 1993.

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

[19] 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.

[20] 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.

[21] 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.

[22] 福田晃. 並列オペレーティングシステム. コロナ社, 1997.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

付録 A

評価プログラムと結果

本章では評価のために作成したプログラムとその結果を示す.

A.1 平均画素法による図形縮小プログラム

A.1.1 ソースコード

```
/**
  @file   reduction.cpp
  @author K.S.
  Create : K.S. 03/01/11 20:24:06
*/
#include <stdio.h>
#include <stdlib.h>

#include <pthread.h>

/// 並列度
#define MAX_PARALLEL 8


#define PIXEL_INITIALIZE {0}
#define PIXEL_SIZE       3

typedef struct{
  int r;
```

```
  int g;
  int b;
}PIXEL;

typedef struct tag_reduction_param{
  /// 必須
  unsigned char *src; /// 縮小する元画像 十分なサイズがあることを前提としている
  int sx,sy;  /// 元画像の x,y サイズ
  unsigned char *dst; /// 縮小した結果の画像 十分なサイズがあることを前提としている
  int dx,dy;  /// 縮小後の x,y

  /// reduction 時設定する
  int *work; /// sx*sy を int ではいっとる

  int index;  /// この処理が何番目か
  int rsx,rsy;
  int rdx,rdy;
  int sline_size; /// src 1line のサイズ
  int dline_size; /// dst 1line のサイズ
}REDUCTION_PARAM;

/// plane の x,y PIXEL に p を設定する
#define PIXEL_DSET(param,x,y,p) {\
  (param)->dst[y*(param)->dline_size + x + 0] = p.r;\
  (param)->dst[y*(param)->dline_size + x + 1] = p.g;\
  (param)->dst[y*(param)->dline_size + x + 2] = p.b;\
}

#define PIXEL_SGET(param,x,y) \
  ((param)->work \
    [(((y)/(param)->rsy) * (param)->sx + (x)/(param)->rsx) * 3])

#define PIXEL_ADD(p,param,x,y) {\
  p->r += PIXEL_SGET(param,x,y,0); \
  p->g += PIXEL_SGET(param,x,y,1); \
  p->b += PIXEL_SGET(param,x,y,2); \
```

```
};

void PIXEL_RGET(REDUCTION_PARAM *param,int dx,int dy,unsigned char *dst){
  int x,y;
  int r = 0;
  int g = 0;
  int b = 0;
  int yy = dy * param->rdy;
  int xx = dx * param->rdx;

  for(y=0;y<param->rdy;y++){
    for(x=0;x<param->rdx;x++){
      int *addr = &PIXEL_SGET(param,xx+x,yy+y);
      r += addr[0];//*addr++;
      g += addr[1];//*addr++;
      b += addr[2];//*addr;
    }
  }
  dst[0] = r / (param->rdy * param->rdx);
  dst[1] = g / (param->rdy * param->rdx);
  dst[2] = b / (param->rdy * param->rdx);
}
th_mutex_t printflock;
void *reduction_child(void *p){
  REDUCTION_PARAM *param = (REDUCTION_PARAM *)p;
  int x,y;
  int start,last;
#if MAX_PARALLEL == 1
  start = 0;
  last  = param->dy;
#else
  start = param->index * param->dy / MAX_PARALLEL;
  last  = (param->index+1) * param->dy / MAX_PARALLEL;
#endif

  for(y=start;y<last;y++){
```

```
    for(x=0;x<param->dx;x++){
      PIXEL_RGET(param,x,y,&param->dst[y*(param)->dline_size + x*3 + 0]);
    }
  }
  return 0;
}

int get_lcm(int m,int n){
  int a,b;

  if(m<n){a = n;b = m;}
  else   {a = m;b = n;}

  while(b!=0){
    int r = a % b;
    a = b;
    b = r;
  }
  return m * n / a;
}

void reduction(REDUCTION_PARAM *param){
  int lcm;

  // パラメータ設定

  lcm = get_lcm(param->dx,param->sx);
  param->rsx = lcm / param->sx;
  param->rdx = lcm / param->dx;
  //printf("%d,",lcm);

  lcm = get_lcm(param->dy,param->sy);
  param->rsy = lcm / param->sy;
  param->rdy = lcm / param->dy;
  //printf("%d\n",lcm);
  {
```

```
    const int padding[] = {0,3,2,1};
    param->sline_size = param->sx * PIXEL_SIZE;
    param->sline_size+= padding[param->sline_size % 4];
    param->dline_size = param->dx * PIXEL_SIZE;
    param->dline_size+= padding[param->dline_size % 4];
  }
  {
    // int で扱うために copy
    int x,y;
    param->work = (int *)malloc(param->sx * param->sy * sizeof(int) * 3);
    if(0)for(y=0;y<param->sy;y++){
      for(x=0;x<param->sx*3;x++){
        param->work[x + y * param->sx*3] =
          param->src[x + y * param->sline_size];
      }
    }
  }
  printf("---");

#if MAX_PARALLEL == 1
  reduction_child(param);
#else
  {
    int i;
    pthread_t th[MAX_PARALLEL];
    REDUCTION_PARAM params[MAX_PARALLEL];
    // 各スレッド用パラメータ設定
    for(i=0;i<MAX_PARALLEL;i++){
      params[i] = *param;
      params[i].index = i;
    }

    for(i=0;i<MAX_PARALLEL;i++){
      if(pthread_create(&th[i],0,reduction_child,&params[i]) != 0){
        printf("error\n");
```

```
      }
    }
    for(i=0;i<MAX_PARALLEL;i++){
      pthread_join(th[i],0);
    }
  }
#endif
}

#define R 2

#define SX 800
#define SY 600
#define SL (800*3)

#define DX 400
#define DY 400
#define DL (400*3)

// #include <windows.h>

unsigned char src[SY*SL] = {0};
unsigned char dst[DY*DL] = {0};

int main(int argc,char *argv[]){
  REDUCTION_PARAM param;
  int dst_size;

  int i;
  //for(i=0;i<SY*SL;i++) src[i] = 1;
  //for(i=0;i<DY*DL;i++) dst[i] = 0x98;

  param.src = &src[0];
  param.sx  = SX;
  param.sy  = SY;
  param.dst = &dst[0];
```

```
  param.dx  = DX;
  param.dy  = DY;

  reduction(&param);

  return 0;
}
```

A.1.2 結果

本研究ではシミュレータ上での評価を行うので，詳細な値を出力することができる．

```
  cycle      fetch      dispatch   finish     retire
> 171545740, 174127680,  55059517,  55059510,  39687289 # AT Num:1
> 142417381, 109902848,  48703160,  48703152,  39687349 # AT Num:2
> 121208078,  75630040,  44924601,  44924593,  39687469 # AT Num:4
> 112600488,  67402096,  43027778,  43027772,  39687709 # AT Num:8

cycle : 総サイクル数     fetch : 総命令フェッチ回数
finish: 総命令実行回数   retire: 総リタイア回数
```

A.2 行列の掛け算を行うプログラム

A.2.1 ソースコード

```
/****
  matrix.c
  $Id: matrix.c,v 1.2 2003/01/08 14:37:18 ko1 Exp $

  Create : K.S. 03/01/05 15:01:55
*/
#include "pthread.h"

typedef struct{
  int n_rows;
  int n_cols;
  int *m;
}matrix_t;

typedef struct{
  int i; // child thread index
  int n; // child thread number
  matrix_t *a,*b,*c; // c = a * b
  int trace_ans;
}mm_params;

#define MREF(mt,row,col) ((mt)->m[(row) * (mt)->n_cols + (col)])

void *matrix_multiply_child(void *param){
  mm_params *p = (mm_params *)param;
  matrix_t *a = p->a;
  matrix_t *b = p->b;
  matrix_t *c = p->c;
  int row,col,j;
  int trace = 0;
```

```
  int quot = c->n_rows / p->n;
  int rem  = c->n_rows % p->n;
  int do_rows = quot + ((p->i < rem) ? 1 : 0);
  int first_r = quot * p->i + ((p->i < rem) ? p->i : rem);
  for(row = first_r ; row < first_r + do_rows; row++){
    for(col=0;col < c->n_cols;col++){
      int sum = 0;
      for(j=0;j<a->n_cols;j++){
        sum += MREF(a,row,j) * MREF(b,j,col);
      }
      MREF(c,row,col) = sum;
      //c->m[row * c->n_cols + col] = sum;
      if(row == col){
        trace += sum;
      }
    }
  }
  p->trace_ans = trace;
  return 0;
}
#ifndef MAX_PARALLEL
#define MAX_PARALLEL 8
#endif

#ifndef MSIZE
#define MSIZE        64
#endif

int matrix_multiply_parent(matrix_t *a,matrix_t *b,matrix_t *c){
  mm_params params[MAX_PARALLEL];
  pthread_t thread[MAX_PARALLEL];

  int i,res;
  int trace = 0;

  if(MAX_PARALLEL == 1){
```

```
    i = 0;
    params[i].i = i;
    params[i].n = MAX_PARALLEL;
    params[i].a = a;
    params[i].b = b;
    params[i].c = c;
    matrix_multiply_child(&params[i]);
    trace += params[i].trace_ans;
  }
  else{
    for(i=0;i<MAX_PARALLEL;i++){
      params[i].i = i;
      params[i].n = MAX_PARALLEL;
      params[i].a = a;
      params[i].b = b;
      params[i].c = c;
      res = pthread_create(&thread[i],0,matrix_multiply_child,(void *)&params[i]);
      if(res != 0){
        printf("error is occur");
      }
    }
    for(i=0;i<MAX_PARALLEL;i++){
      pthread_join(thread[i],0);
      trace += params[i].trace_ans;
    }
  }
  return trace;
}


extern int a_body[]; // 計算する行列の中身は別ファイルで定義してある
extern int b_body[]; // 計算する行列の中身は別ファイルで定義してある

main(){
  int c_body[MSIZE*MSIZE];
  matrix_t a = {MSIZE,MSIZE,&a_body[0],};
```

```
  matrix_t b = {MSIZE,MSIZE,&b_body[0],};
  matrix_t c = {MSIZE,MSIZE,&c_body[0],};

  matrix_multiply_parent(&a,&b,&c);

  // 結果表示するためには 1 にする
  if(0){
    int i,j;
    for(i=0;i<MSIZE;i++){
      printf("    ");
      for(j=0;j<MSIZE;j++){
        printf("%d,",c.m[i * c.n_cols + j]);
      }
      printf("\n");
    }
  }

  return 0;
}
```

A.2.2 結果

```
  cycle      fetch      dispatch   finish     retire
> 223099323, 543206368, 153694555, 153694553, 152638266 # AT Num:1
> 221262527, 273190776, 153434819, 153434817, 152638334 # AT Num:2
> 219927016, 205019556, 152829087, 152829086, 152638470 # AT Num:4
> 219812759, 192139438, 152695282, 152695278, 152638498 # AT Num:8

cycle : 総サイクル数     fetch : 総命令フェッチ回数
finish: 総命令実行回数   retire: 総リタイア回数
```