ユーザレベルスレッドライブラリPPLにおける柔軟なスケジューリング機構

北口修一[†,☆] 松浦慎二[†,☆☆]
最所圭三[†] 福田晃[†]

ユーザレベルスレッドライブラリ上で柔軟なスケジューリングを行うことができる機構について論じる．これまで，並行/並列処理の実行単位であるスレッドを，ユーザレベルで提供するスレッドライブラリが多く実現されているが，扱えるスケジューリングポリシは限られている．しかし，アプリケーションが多様化している状況を考えると，新たなポリシを持つスケジューラの追加，スケジューラ間の柔軟な調整，などの機能が必要になる．本論文では，複数のポリシを同時に扱うために，スケジューラをオブジェクトとして扱うポリシオブジェクトモデルを提案し，ポリシオブジェクトを管理するポリシスケジューラを導入した．さらに，スケジューラを作成するための環境を提供することにより，ユーザがスケジューラを容易に作成できるようにした．提案しているスケジューリング機構を，開発中のユーザレベルスレッドライブラリPPL上に実装し，評価を行った．

Flexible Scheduling Mechanism on User-level Thread Library PPL

Shuichi Kitaguchi,[†,☆] Shinji Matsuura,[†,☆☆] Keizo Saisho[†]
and Akira Fukuda[†]

A flexible scheduling mechanism for user-level thread library is discussed. Although many thread libraries which schedule threads at user level are implemented, they can operate few scheduling policies. Since new applications having new scheduling policy appear, a scheduling mechanism, which can take new scheduling policy in itself and coordinate scheduling policies flexibly, would be needed. In this paper, the policy object model is proposed and the policy scheduler is introduced. The policy object model regards a scheduler as an object and the policy scheduler coordinates policy objects. Moreover, by providing users an environment for making schedulers, they can make scheduler by themselves easily. Proposed mechanism is applied on user-level thread library PPL that authors are now implementing and evaluated.

1. はじめに

現在，並列処理の実行単位としてスレッドが一般に使用されるようになってきており，Mach[1]などの多くのOSでもスレッドの実行環境が提供されている．スレッドは，プロセスとは異なり，メモリ空間やファイルデスクリプタなどのリソースを共有しているので，基本的な操作である生成/消滅/切替えの際にメモリ空間の生成/消滅/切替えを行う必要がなく，これらの処理のオーバヘッドを小さくすることができる．しかし，OSが提供するスレッドの実行環境では，スレッド操作を行うたびにシステムコールを発行するため，メモリ空間の切替えが発生し，それにともなうオーバヘッドを生じる．このため，OSが提供するスレッドやプロセスなどを仮想プロセッサと見なし，その上でユーザのスレッドを動作させるユーザレベルスレッドライブラリの研究が数多くなされている．代表的なものとして，MachのCthreads[2]，MuellarによるSunOS専用のスレッドライブラリ[3]，安倍らのPortable Thread Library[4]など多くの研究がある．これらのユーザレベルスレッドライブラリを並列性と移植性の両方から見ると，CthreadsはMachが提供する並列性を有するスレッドを仮想プロセッサとしているので並列性を有するが，Machに依存するため移植性がない．MuellarのスレッドライブラリはSunOSに限定することで高速性を追求しており，移植性や並列性はない．また，

† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
☆ 現在，富士通株式会社
Presently with Fujitsu Limited
☆☆ 現在，シャープ株式会社
Presently with Sharp Corporation

PTL は BSD UNIX であればほとんど変更することなく移植でき，非常に移植性に優れているが，1つのUNIXプロセスを仮想プロセッサとしているので，並列性がない．このため，我々は並列性と移植性を考慮したParallel Pthread Library（PPL）の開発を行っている[5),6)]．

PPLはメモリ空間を共有する複数の仮想プロセッサを扱うことができ，それらが並列に動く環境であれば，スレッドも並列に動作させることができるようになっている．また，様々なシステム上への移植を容易にするため，ライブラリ内部を仮想プロセッサに依存する部分とそれ以外の部分に徹底してモジュール化している．さらに，ユーザレベルでの移植性を考慮して，標準的なスレッドのインタフェースであるIEEE POSIXのpthreadインタフェース[7)]を採用している．この並列性と移植性の両方を兼ね備える点が，PPLの特徴である．PPLと同様に並列性と移植性を考慮したユーザレベルスレッドライブラリとして，最近，小熊らのSMP型計算機を活用する軽量プロセス・ライブラリ[8)]が開発されている．

pthreadインタフェースではスケジューリングポリシとしてFIFO，ラウンドロビン（以下，RR），OTHER（スレッドライブラリの実装に依存）の3つのみ定義されており，旧バージョンのPPLを含め，現在のユーザレベルスレッドライブラリもその仕様に従って実装されている．しかし，アプリケーションが多様化している状況を考えると，ユーザレベルスレッドライブラリの開発者が想定していないスケジューリングポリシが必要になってくると考えられる．たとえば，マルチメディア処理では実時間性が要求される．アプリケーションが決まっていれば，OTHERで実現することも可能であるが，それだと用途が限られてしまう．そこで，本研究では，1) アプリケーションレベルで新たなスケジューラを作成・追加でき，2) 複数のスケジューラを混在させ，それらの間の調整をアプリケーションレベルで行うことができる，柔軟なスケジューラの実現を目指すことにした．この柔軟なスケジューラを実現できれば，異なるスケジューリングポリシを持つスレッドを混在させたい場合，混在させるポリシでスケジューリングするスケジューラを独立して作成・追加し，追加したスケジューラ間の調整をアプリケーションレベルで行うことができる．

アプリケーションレベルでスケジューラ間の調整を行うために，スケジューラをオブジェクトとして扱うポリシオブジェクトモデルを提案し，それらを管理するためのポリシスケジューラを導入した．スケジューリングのポリシは，レディキューへの連結およびレディキューからの選択の方法で決まるので，この2つの操作をポリシオブジェクトの基本機能とした．また，ポリシスケジューラがスケジューラ間の調整を行うための情報（優先度など）もポリシオブジェクトに含めている．この情報をアプリケーションレベルで操作することによりスケジューラ間の調整を行うことができる．より柔軟なスケジューラ間の調整を行うことができるように，ポリシスケジューラ自身もアプリケーションレベルで定義できるようにした．その他の機能として，追加したポリシをスレッドの属性に設定するための操作も加えている．ポリシオブジェクトにより，アプリケーションレベルで新たなスケジューラを追加する際のインタフェースを，非常に単純なものにできた．さらに，アプリケーションレベルでスケジューラの作成を支援するための環境も提供している．

上記の方針で設計したスケジューリング機構のプロトタイプを実装し評価することで，提案しているスケジューリング機構が有効であることを確認した．

本論文では，2章で対象のユーザレベルスレッドライブラリPPLの概要を述べる．3章で柔軟なスケジューリング機構の概念および設計方針を述べる．4章で実装を述べ，5章で評価を行う．

2. PPLの概要

本章では，本論文で提案する柔軟なスケジューラを実装するPPLの並列性と移植性について述べる．

2.1 並列性の実現

PPLは，スレッドの制御をユーザ空間で行うユーザレベルスレッドライブラリである．PPLは，**図1**に示すように，OSが与えるスレッドやプロセスを仮想プロセッサ（VP）と見なして，ユーザレベルでスレッド（TH）と仮想プロセッサとの対応付けを行うものである．仮想プロセッサと物理プロセッサ（P）との対応付けはOSが行う．ここで，メモリ空間を共有す

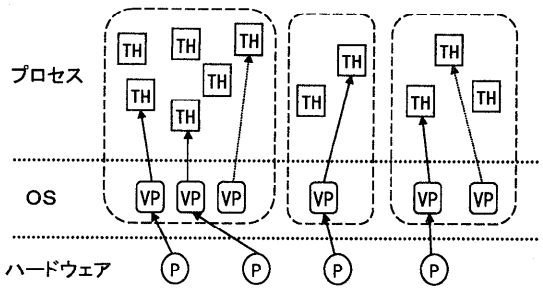

図1 PPLの実行モデル
Fig. 1 Execution model of PPL.

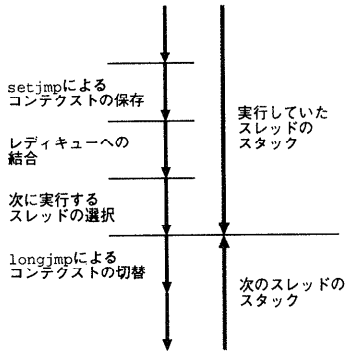

図2 *setjmp*, *longjmp* の単純な組合せによるコンテクストスイッチ
Fig. 2 Context switch using simple combination of *setjmp* and *longjmp*.

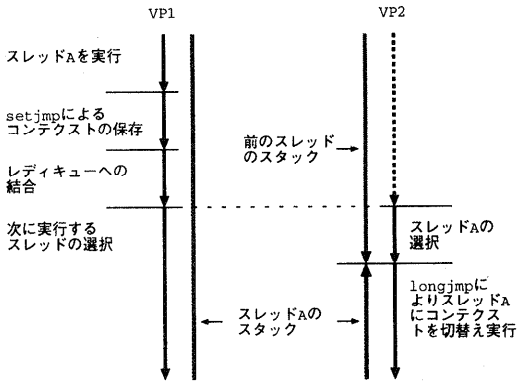

図3 マルチプロセッサシステムでの問題点
Fig. 3 Problem on multiprocessor systems.

る仮想プロセッサが複数与えられ，それらが実際に並列に実行できるなら，PPL 上でスレッドを実際に並列に実行させることができる．

多くのユーザレベルスレッドライブラリにおいては，多田ら[9]と同様に *setjmp* と *longjmp* を単純に組み合わせてコンテクストスイッチを行うコルーチン方式が用いられている．しかし，PPL では，仮想プロセッサが実際に並列に実行するため，*setjmp* と *longjmp* の単純な組合せでは，コンテクストスイッチを実現できない．

setjmp と *longjmp* を単純に組み合わせて実現するコンテクストスイッチの概念を**図2**に示す．実行していたスレッドのスタックを使用してスケジューリングが行われている．この方法は，仮想プロセッサが1つしかない場合は問題にならないが，複数の仮想プロセッサを持つ場合は2つの仮想プロセッサで同じスタックを使用するという問題が発生する可能性を持つ．その例を**図3**に示す．アイドル仮想プロセッサ VP2 が存在するときに，これまで実行していた仮想プロセッサ VP1 がスレッド A をレディキューに連結すると，その瞬間 VP2 がスレッド A を取り出して実行することになり，VP2 のスタックはスレッド A のスタックになる．VP1 は，スレッド A のスタックを用いてスケジューリングを行っているので，スレッド A のスタックが VP1 と VP2 の両方で使用される．

これを防ぐ方法として，スケジューラに入ることができるプロセッサを1つに制限することが考えられるが，並列性が犠牲になる．そこで，PPL では，スレッドと独立して，各仮想プロセッサにスケジューリングのときに使用する固有のスタックを与えるようにした．**図4**に示すように，スレッドのコンテクストを保存した後，スケジューラ用のスタックに切り替え，それを

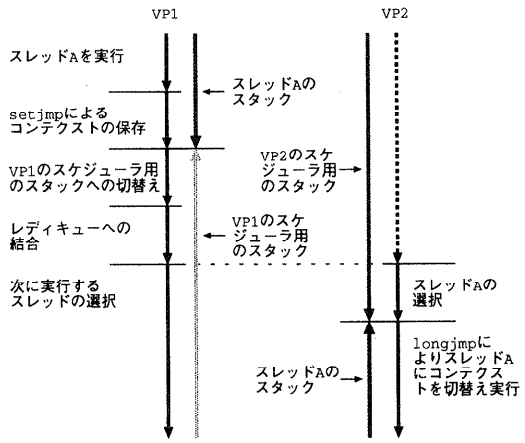

図4 スタック共有問題の解決法
Fig. 4 The method to solve stack sharing problem.

用いてスケジューリングを行うようにした．これにより異なった仮想プロセッサが同じスタックを使用することはなくなる．ただし，仮想プロセッサごとにスケジューラ用のスタックを与えなければならないので，スケジューラは，どの仮想プロセッサで実行されているかを検出し，対応するスタックを選択しなければならない．選択したスタックへの切替えは，*longjmp* を用いて実現できる．

2.2 移植性の実現

PPL は，移植性として，ユーザレベルの移植性とスレッドライブラリ自身の移植性を考慮している．このため，**図5**に示すような構成にした．

(1) アプリケーションの移植性
ユーザに対して共通のインタフェースを提供するために，PPL ではスレッドインタフェースの事実上の標準となっている IEEE POSIX によって標準化された pthread インタフェース

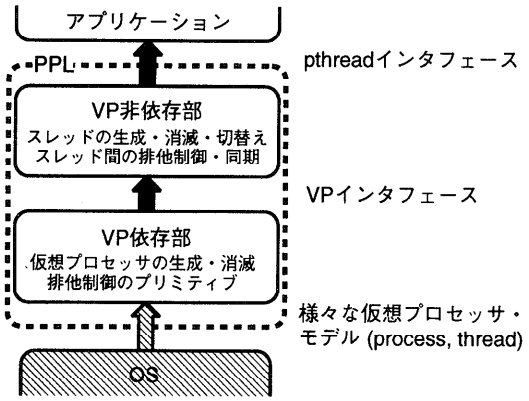

図5 PPLの構成
Fig. 5 Structure of PPL.

を採用した．このインタフェースは，PTLやLinux[11]上のスレッドライブラリであるLinuxThreads[10]でも採用されている．したがって，pthreadインタフェースを用いて記述されたスレッドプログラムをPPL上で，変更を加えることなく動作させることが可能となる．旧バージョンPPLでは，pthreadインタフェースとして，規格案であったPOSIX 1003.4a[12]を採用していたが，現在開発中のバージョンでは，正式に公開されたPOSIX 1003.1c[7]を採用している．

(2) スレッドライブラリ自身の移植性
PPL自身の種々のOSへの移植を容易にするために，ライブラリの内部を，OSに依存する部分（VP依存部）と依存しない部分（VP非依存部）とに徹底的に分離した．VP依存部は，図5に示すように，与えられる仮想プロセッサの種類や排他制御のプリミティブのようにOSに依存する処理を含む．VP非依存部では，VP依存部によって仮想化されたプロセッサや排他制御のプリミティブを用いて，スレッドの生成/消滅/切替え，スレッド間の排他制御・同期を実現する．また，メモリ操作やロック機構といった機能ごとのモジュール化を図ることによって，移植の際の変更箇所を明確にするとともに，変更も最小限に抑えられるようにしている．

3. 柔軟なスケジューラ

本章においては，柔軟なスケジューラの設計および構成について述べる．

3.1 柔軟なスケジューラの設計方針

pthreadインタフェースではスケジューリングポリシとしてFIFO，RR，OTHERの3つが定義されているが，それらのスケジューリングポリシを混在して使用する場合の動作は規定されていない．また，OTHERは，スレッドライブラリの実装に依存するため，そのポリシがアプリケーションに適しているかを保証できない．たとえば，PTLでは，OTHERは，優先度に比例したCPU時間の配分を行うが，FIFOとRRより優先度が低いため，FIFOかRRのスレッドが存在するとこの機能は使用できない．また，LinuxThreadsでは，スレッドはメモリを共有するプロセスに1対1で対応しており，生成したスレッドの個数のプロセスを使用し，スケジューリングはカーネル依存になる．

そこで，我々は，より柔軟なスケジューリングを行うことができるように，以下の2つの要件に基づいて柔軟なスケジューラの設計を行った．

(1) アプリケーションレベルで新たなスケジューラの作成・追加することができること．

(2) 複数のスケジューラを混在させ，それらの間の調整をアプリケーションレベルで行うことができること．
なお，混在させるスケジューラには，pthreadインタフェースで規定されているスケジューリングポリシでスケジューリングを行う3つのスケジューラに加えて，ユーザが追加するスケジューラがある．

3.1.1 アプリケーションレベルでのスケジューラの作成・追加

アプリケーションに適したスケジューリングポリシは，ユーザが最も理解していると考え，ユーザにスケジューラを作成・追加できるようにすることが最も適した方法であると考えた．この際，以下の2点を考慮しなければならない．

(1) ユーザが作成する部分を可能な限り小さくする．
スケジューラは多くの機構から構成されるが，ポリシに依存する部分はそれほど大きくない．そこで，ポリシに依存する部分を他の部分から分離し，その部分だけ作成することで，スケジューリングポリシを実現できるようにする．これにより，ユーザが作成しなければならない箇所を小さくする．

(2) PPLの内部構造をできる限り隠蔽する．
スケジューラの作成にPPLの内部構造が影響してしまうのでは，ユーザにPPLを理解するという作業を課すことになる．そのため，PPL

の内部構造を知らなくてもスケジューラを作成できるようにしなければならない．

まず，(1)について議論する．スケジューラの仕事の主なものは，スレッドのコンテクストスイッチである．スレッドのコンテクストスイッチの手順は，a) 現在実行中のスレッドのコンテクストの保存，b) レディキューへの連結，c) 次に実行するスレッドの選択，d) 選択したスレッドのコンテクストの復帰，の4つの手順で行われる．このうち，レディキューを操作するb)とc)がスケジューリングのポリシを決定する．そこで，この部分を分離し，レディキューを各スケジューラに持たせ，他の部分とのインタフェースとしてレディキューへの連結とレディキューからの選択の2つのみとした．また，レディキューへの連結は，コンテクストスイッチ以外にロックや子スレッドの終了を待っているスレッドが実行可能になるときも発生するので，この切り分けはこれらにも適している．

スケジューラをオブジェクトの形で与えるようにし，これらを表の形で管理することによりスケジューラの追加を実現する．このオブジェクトを我々はポリシオブジェクトと呼び，表をポリシオブジェクトテーブルと呼んでいる．このポリシオブジェクトテーブルにポリシオブジェクトを加えることにより，スケジューラの追加を実現する．

次に，(2)について議論する．レディキューの操作は，スレッドの情報を持つスレッドコントロールブロックの構造に大きく影響される．また，スレッドコントロールブロックの構造はPPLの内部構造に大きく影響される．このため，レディキューを操作するには，PPLの内部構造を理解しなければならないことになる．しかし，レディキューの純粋な操作はポリシが変わっても共通化できる．たとえば，優先度方式の場合，レディキューに連結する際に，先頭の要素から調べて挿入する場所を探し，そこに連結することになる．この操作は，キューの先頭のスレッドへのポインタを得る，指定したスレッドの優先度を調べる，指定したスレッドの次の要素へのポインタを得る，指定したスレッドの次に挿入する，の4つの操作で実現できる．そこで，これらを基本関数群として提供することによりPPLの内部構造を隠蔽することにした．

3.1.2 アプリケーションレベルでのスケジューラ間の調整

複数のスケジューラを混在させる場合，スケジューラ間の調整が必要になる．この調整がアプリケーションの意図するものと異なると，アプリケーションがうまく動かないことが生じる．たとえばFIFOとRRを同時に使用する場合，すべての仮想プロセッサにFIFOのスレッドが割り当てられるとRRのスレッドを動かすことができなくなる．そのため，アプリケーションレベルでのスケジューラ間の調整が必要になる．PPLでは複数の仮想プロセッサを持つことができるので，スケジューラ間を調整する方法として以下の2つが考えられる．

(1) スケジューラごとに仮想プロセッサ群を割り当てる
この方法は，割り当てられた仮想プロセッサを用いて各スケジューラが独立してスケジューリングを行うものである．つまり，ポリシごとにプロセッサ資源を分割するものである．このため，各スケジューラを独立して動作させることができる，スケジューリングのオーバヘッドが増加しない，などの利点を持つ．しかし，割り当てられた仮想プロセッサ以上の実行可能なスレッドがない場合にアイドル状態になる仮想プロセッサが発生する，同時に使用するスケジューラの個数以上の仮想プロセッサを確保できない場合に使用できない，などの欠点を持つ．

(2) 新たなスレッドを実行するときに調整する
この方法は，次に実行するスレッドをどのスケジューラを用いて選択するかを決めるものである．そのため，スケジューラの選択のためのオーバヘッドが生じる．しかし，ポリシ間の調整を動的に行うことができるので，より柔軟なスケジューリングが可能となる，仮想プロセッサ数が同時に使用するスケジューラの個数より少なくても使用することができる，などの利点を持つ．

本研究では，できる限り柔軟なスケジューリング機構を提供するために，方式(2)を採用することにした．このため，スケジューラ間の調整を行うスケジューラを導入する．これ以降，スレッドのスケジューリングを行うスケジューラをスレッドスケジューラ，スレッドスケジューラ間の調整を行うスケジューラをポリシスケジューラと呼ぶ．ポリシスケジューラは全体で1つ存在する．

スレッドスケジューラ間を調整するためには，スレッドスケジューラごとにポリシを決めるパラメータが必要になる．必要とするパラメータの個数やどのようなパラメータが必要であるかは，ポリシスケジューラがどのような基準でスレッドスケジューラ間を調整するかに依存する．たとえば，スレッドスケジューラに優先度を与える方式，各スレッドスケジューラが必要と

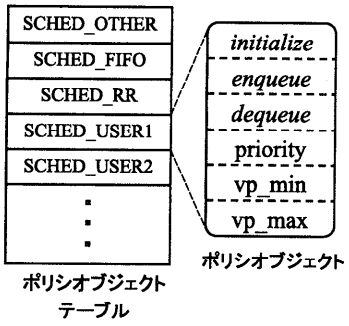

図6 ポリシオブジェクトテーブル
Fig. 6 Policy object table.

する最小と最大の仮想プロセッサ数を指定する方式，などが考えられる．パラメータをアプリケーションレベルで設定できるようにすることで，アプリケーションレベルでのスレッドスケジューラ間の調整をできるようにする．しかし，パラメータの調整だけでは限界があるので，ポリシスケジューラ自体もアプリケーションレベルで変更できるようにする．このために，ポリシスケジューラを呼び出すためのインタフェースを決めなければならない．3.1.1 項でも述べたように，ポリシはレディキューへの連結と次に実行するスレッドの選択で決まるので，ポリシスケジューラとのインタフェースもこの 2 つに限定できる．スレッドスケジューラとの違いは，対象がスケジューラであるという点である．

3.2 柔軟なスケジューラの構成

全体の構成を述べる前に，スレッドスケジューラとポリシスケジューラとのインタフェースである，ポリシオブジェクトについて述べる．

3.2.1 ポリシオブジェクトテーブル

3.1 節で示した設計方針に従ったスケジューリング機構を実現するために，**図 6** に示すポリシオブジェクトテーブルという機構を導入する．ポリシオブジェクトとは，3.1.1 項で述べたレディキューへの連結関数（*enqueue*）とレディキューからの選択関数（*dequeue*），ポリシを決めるパラメータ，およびスケジューリングするスレッドの属性を初期化する関数（*initialize*）からなる．*enqueue*, *dequeue*, *initialize* は，実際に処理する関数へのポインタになっている．また，パラメータはポリシスケジューラの調整の方法によって内容が変わる．図では，実装したプロトタイプのものになっている．詳しくは，3.2.3 項で述べる．

ポリシオブジェクトを表の形でまとめたものがポリシオブジェクトテーブルである．ポリシオブジェクトテーブルには，pthread インタフェースで定義されている 3 つのスケジューリングポリシのほかに，新たなスレッドスケジューラを追加するための枠を用意している．

3.2.2 コンテクストスイッチに関連する部分

図 7 に，柔軟なスケジューリング機構の構成およびスレッドの流れを示す．図に示すように，スケジューリング機構は，コンテクストスイッチに関連する部分，ポリシスケジューラ，スレッドスケジューラの 3 層で構成される．

本機構における処理の流れは以下のとおりである．

(1) スレッドからの自発的なプロセッサの解放やタイマ割込みによって発生する，スケジューリング要求がコンテクストスイッチに関連する部分に入る（①）．この部分では，*setjmp* によってコンテクストを保存し，スケジューラ用のスタックに切り替えた後，ポリシスケジューラにコンテクストを保存したスレッドを渡す（②，ⓐ）．

(2) ポリシスケジューラは，渡されたスレッドのポリシに対応するスレッドスケジューラに，そのスレッドを渡す（③，ⓑ）．

(3) スレッドを渡されたスレッドスケジューラは，渡されたスレッドを自分のレディキューに連結する（④）．これでスレッドのレディキューへの連結が終了する．

(4) 次に，コンテクストスイッチに関連する部分は，次に実行すべきスレッドを選択するためにポリシスケジューラに依頼する（ⓒ）．

(5) ポリシスケジューラは，次に実行するスレッドのポリシを決定し，そのポリシを持つスレッドスケジューラに依頼する（ⓓ）．

(6) 依頼を受けたスレッドスケジューラは，自分のレディキューからスレッドを取り出し（⑤），ポリシスケジューラに渡す（⑥）．このとき，ポリシに依存した特殊な設定があればその設定を行う．

(7) ポリシスケジューラは，渡されたスレッドをコンテクストスイッチに関連する部分に渡す（⑦）．

(8) コンテクストスイッチに関連する部分では，渡されたスレッドの属性に応じた設定を行った後，そのスレッドのコンテクストを復帰する（⑧）．スレッドの属性に応じた設定として，RR の場合のタイムクオンタムの設定などがある．

括弧内の丸で囲まれた数字はスレッドの流れを，アルファベットは制御の流れを示し，図 7 の丸で囲まれた数字とアルファベットと対応している．また，現在の実装では，連結のための関数と選択のための関数の名

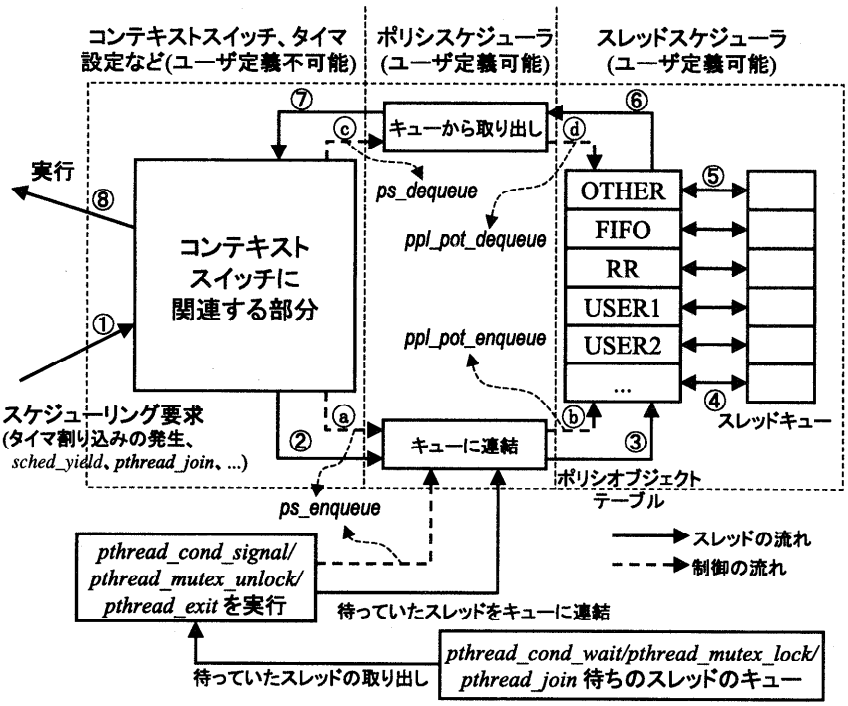

図7 スケジューリング機構の構成
Fig. 7 Structure of scheduling mechanism.

前を，*ps_enqueue* および *ps_dequeue* としているが，名前だけの問題なので，変更は容易である．

3.2.3 ポリシスケジューラ

ポリシスケジューラは，3.2.2 項で述べた 2 つの関数（*ps_enqueue* および *ps_dequeue*）によって呼び出される．

ps_enqueue 各スレッドに属性としてポリシが付属しているので，使用するスレッドスケジューラは一意に定まる．スレッドのポリシに従ってスレッドスケジューラを選択し，スレッドスケジューラとのインタフェースである *ppl_pot_enqueue* 関数を経由して，該当するスレッドスケジューラのレディキューに連結する．*ps_enqueue* 関数は，コンテクストスイッチに関連する部分からだけでなく，待ち状態になっていたスレッドが実行可能になるときも呼び出される．

ps_dequeue この関数は，コンテクストスイッチに関連する部分からのみ呼び出され，次に実行すべきスレッドを選択する．ポリシスケジューラでは，何らかの基準でスレッドスケジューラを選択し，スレッドスケジューラとのインタフェースである *ppl_pot_dequeue* 関数を経由して，該当するスレッドスケジューラを呼び出す．スレッドスケジューラの選択の方法は実装依存になり，アプリケーションレベルで変更するのはこの部分になる．実装したプロトタイプで採用した方式は，4.2 節で述べる．

3.2.4 スレッドスケジューラ

スレッドスケジューラは，内部にレディキューを持ち，実際にスレッドを管理する．3.2.1 項で述べたように，初期化関数，連結関数，選択関数の 3 つを外部に公開している．スレッドスケジューラの作成のための基本関数群は，キュー操作，スレッドの属性操作，ポリシオブジェクト操作の 3 つのグループに分かれる．たとえば，キュー操作の関数として，

- pthread_t *ppl_get_queue_head*(pthread_t queue)
  引数で指定したキューの先頭スレッドを取得する．ただし，キューが空の場合は NULL が返る．
- pthread_t *ppl_get_next_thread*(pthread_t thread)
  引数で指定したスレッドの，1 つ後に位置するスレッドを取得する．ただし，指定したスレッドが末尾だった場合，NULL が返る．
- void *ppl_insert_thread*(pthread_t point,
  pthread_t thread)
  第 1 引数のスレッドの後ろに，第 2 引数のスレッドを実際に連結する．

などがある．また，スレッドの属性操作およびポリシオブジェクトにスケジューリング関数を登録するために，

- void *ppl_set_thread_policy*(pthread_t thread, int policy)
  第1引数で指定するスレッドに，第2引数で指定するポリシ属性を設定する（標準のpthreadでは，このようなポリシの動的変更はサポートされない）．
- void *ppl_pot_set_funcs*(int policy, int (*init)(pthread_attr_t*), int (*enq)(pthread_t), pthread_t (*deq)(void))
  ポリシオブジェクトテーブル上の，第1引数で指定するポリシに，作成した *initialize*, *enqueue*, *dequeue* をまとめて登録する．

などを用意した．

4. 実　　装

本章においては，3章の設計方針および構成に基づいて実装したプロトタイプの，実装依存に関する部分について説明する．

4.1 実装環境

PPLを以下の環境に移植するとともに，柔軟なスケジューリング機構を追加した．

- CPU：PentiumII（300 MHz）
- プロセッサ数：2個
- OS：Linux 2.1.108

PPLの概要で述べたVP依存部では，仮想プロセッサとしてLinuxが提供するメモリ空間を共有するプロセスを与えた．排他制御に関しては，Test & Set命令をアセンブリ言語を用いて実装した．この部分は，非常に短くアセンブリ言語を用いても，移植性にはほとんど影響しない．

4.2 ポリシスケジューラ

2種類のポリシスケジューラを実装した．1つは，各スレッドスケジューラに順々に割り当てるもので，割当ての機会は均等になる．この方法は単純でオーバヘッドは少ないが，FIFOなどの中断しないポリシが入っており，そのポリシを持つスレッドが仮想プロセッサ数より多く，しかも長時間実行される場合，最終的に，すべての仮想プロセッサが，それらのスレッドに占有され，他のポリシのスレッドを動作できなくなるという欠点を持つ．もう1つの方法は，スレッドスケジューラの優先度と必要とする仮想プロセッサ数を用いるものである．仮想プロセッサ数をパラメータとして用いる理由は，単純な優先度を用いる方法よりも柔軟なスケジューリングを可能とするためである．そのため，図6に示す3つのパラメータpriority，vp_min，vp_maxを用意した．これらのパラメータをアプリケーションから調整できるようにしている．以下にスレッドスケジューラの選択のアルゴリズムを示す．

- 各スレッドスケジューラに少なくともvp_min個の仮想プロセッサが割り当てられるようにする．したがって，各スレッドスケジューラのvp_minの合計以上の仮想プロセッサが存在すれば，すべてのスレッドスケジューラの最低要求を満たすことができる．
- 各スレッドスケジューラのpriorityの値によって以下の順序で仮想プロセッサの割当てを行う．
  - SCHED_PRIO_HIGH　この優先度を持つスレッドスケジューラのレディキューにスレッドがあり，かつ，そのスレッドスケジューラに割り当てられている仮想プロセッサがvp_maxより少ない場合は，vp_maxに達するまで仮想プロセッサを割り当てる．
  - SCHED_PRIO_DEFAULT　上記の割当てを行った後に，使用されていない仮想プロセッサが存在すれば，この優先度を持つスレッドスケジューラに対して，上記と同じ方法によって仮想プロセッサを割り当てる．
  - SCHED_PRIO_LOW　上記2つの割当てを行った後に，使用されていない仮想プロセッサが存在すれば，この優先度を持つスレッドスケジューラに対して，上記と同じ方法によって仮想プロセッサを割り当てる．

このアルゴリズムにより，各スレッドスケジューラに最低限の仮想プロセッサを割り当てることができる．さらに，他のスレッドスケジューラより優先して実行したいポリシ（SCHED_PRIO_HIGH）を持つスレッドスケジューラに優先的に仮想プロセッサを与えることができる．

3.1.2項で述べたようにポリシスケジューラは全体で1つしか持てないので，どちらを使用するかを選択しなければならない．現在のところ，ポリシスケジューラごとにライブラリを用意し，プログラムをリンクするときにライブラリを選択することで対応している．将来的には，アプリケーションの中から選択できるようにする予定である．

5. 評　　価

本章では，柔軟なスケジューラを実装したPPLにおけるスレッドスケジューラ間の調整の効果を調べた．

5.1 実験の方法

FIFOとRRの2つのポリシを持つスレッドを同時

に実行させ，4.2節の後半で説明したポリシスケジューラを用いてスレッドスケジューラ間の調整を行った．以下にテストプログラムの概要を示す．

- 2CPUシステム上で3つの仮想プロセッサを用いる．
- 実行するスレッド

  RRのポリシを持つスレッド： プロセッサ1個を占有して実行したときに，50ミリ秒間隔で現在時刻を測定するループを200回繰り返す．このスレッドを10個生成する．したがって，これらのスレッドを1つのCPUを占有して実行したときの実行時間は100秒になる．

  FIFOのポリシを持つスレッド： プロセッサ1個を占有して実行したときの実行時間が2秒かかるスレッドで，開始時刻と終了時刻を測定する．このスレッドを15個生成する．したがって，これらのスレッドを1つのCPUを占有して実行したときの実行時間は30秒になる．

  全スレッドを1つのCPUを占有して実行したときの実行時間は130秒になる．
- スケジューラ間の調整の手順
  (1) プログラムの開始時にFIFOとRRのスレッドスケジューラのvp_minをそれぞれ2と1に設定する．仮想プロセッサ数が3なので，FIFOとRRのスレッド全体に与えられる仮想プロセッサ数はそれぞれ2個と1個になる．
  (2) 7番目のFIFOのポリシを持つスレッドは実行開始時に，FIFOとRRのスレッドスケジューラのvp_minをそれぞれ1と2に変更する．この変更により，次のFIFOのコンテクストスイッチから，FIFOとRRのスレッド全体に与えられる仮想プロセッサ数はそれぞれ1個と2個になる．
- 比較のために同じスレッド群をLinuxThreadsを用いて実行する．

5.2 実験結果

図8と図9に結果を示す．図中のグラフは，各スレッドに与えられたCPU時間を示す．横軸は測定した時刻をプログラム開始時刻からの相対時間に変換したものを示し，左側の縦軸は各スレッドに与えられたCPU時間，右側の縦軸はFIFOとRRのスレッドに与えられらCPU時間の合計を示す．PPL，LinuxThreadsのどちらの場合も経過時間が65秒ですべてのスレッドが終了しており，この時間は1個のCPUで実行したときにかかる時間の半分になっており，2つのプロセッサで並列に実行されたことが分かる．また，スレッドの粒度が非常に大きいので，どちらの場合もコンテクストスイッチの影響が出ていない．

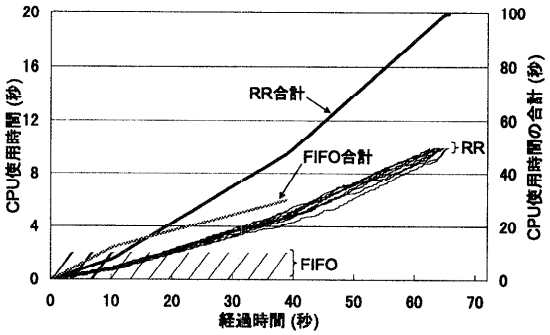

図8 PPLにおいてスレッドに与えられるCPU時間
Fig. 8 CPU time assigned to threads with PPL.

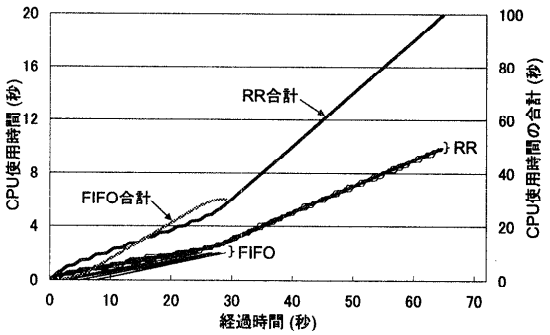

図9 LinuxThreadsにおいてスレッドに与えられるCPU時間
Fig. 9 CPU time assigned to threads with LinuxThreads.

全体の実行時間に関しては，PPL，LinuxThreadsでまったく差は出なかったが，個々のスレッドに関しては動作が大きく異なっている．FIFOのスレッドを見るとPPLの場合は1つの仮想プロセッサ上では1つずつ実行されていることが分かる．これに対して，LinuxThreadsでは，開始時刻は異なるが，いったん実行が開始されると，FIFOに限らずすべてのスレッドに均等にCPU時間が割り当てられていることが分かる．PPLのRRのスレッドを見ると，LinuxThreadsのRRのスレッドと異なり，各スレッドに与えられるCPU時間のばらつきが大きく，最大で約2割のばらつきを生じている．これは，仮想プロセッサに与えられるCPU時間のばらつきが影響しているものと思われる．PPLではRRのスレッドは，実時間で200ミリ秒のタイムクオンタムが与えられるが，途中で非同期のディスク書き込みなどが発生すると，1つのタイムクオンタムで与えられるCPU時間が200ミリ秒よ

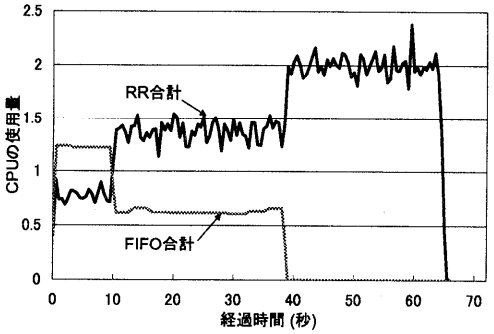

図10 PPLにおけるFIFOとRRに与えられるCPU資源の推移

Fig. 10 Transition of assigned CPU resource to FIFO and RR threads with PPL.

り短くなる．

次にPPLに関して，スケジューラ間の調整を行った効果を調べてみる．7番目のFIFOのスレッドの開始直後にFIFOのポリシを持つスレッドに1個の仮想プロセッサを使用するように変えているので，6番目のFIFOのスレッドまでは同時に2つ実行されているが，7番目のFIFOのスレッド以降は1つのFIFOのスレッドのみ実行されていることが分かる．また，これにともない，RRに与えられるCPU時間の増加率が上がっている．これを確認するために，FIFOとRRのスレッドについて，単位時間あたりに与えられたCPU時間の合計をCPU台数で正規化した（図10）．図中のRRに関するグラフのゆれは，タイムクオンタム内に与えられるCPU時間のゆれが影響している．6番目のFIFOのスレッドの終了（9.5秒）まではFIFOとRRに対して約2:1のCPU資源が与えられている．7番目のFIFOのスレッドから最後のFIFOのスレッドが終了（39秒）するまでは，その比率が約1:2に変わっており，パラメータを変更した効果が出ている．すべてのFIFOのスレッドが終了した後は，RRのスレッドに物理CPUの台数である2個分のCPU資源が与えられている．

以上の結果より，実装したポリシスケジューラが，2つのスレッドスケジューラ間を，必要最小の仮想プロセッサ数を与えることにより調整していることを示すことができた．

6. まとめ

本論文では，スケジューリングポリシをアプリケーションレベルで追加でき，スケジューリングポリシ間の調整をアプリケーションレベルで行うことができる柔軟なスケジューラを提案した．さらに，開発中のユーザレベルスレッドライブラリPPL上にプロトタイプを実装し，柔軟なスケジューラの有効性を実験により確認した．

アプリケーションレベルで自分自身をスケジューリングするのに適したスレッドスケジューラを作成することができるように，スレッドスケジューラの構成をポリシを決定するレディキューの操作に関する部分と，それ以外の部分に分離した．これによりポリシに関する部分の記述量を削減できた．また，構造が簡単になるので記述も容易になった．さらに，基本関数群を用意することにより，PPLの内部情報を隠蔽するとともにレディキュー操作の記述を容易にした．

複数のスケジューリングポリシを同時に扱うことができるように，ポリシオブジェクトテーブルを導入した．これにより，アプリケーションレベルで作成したスレッドスケジューラを取り込むことができるようになった．さらに，スレッドスケジューラ間の調整を行うスケジューラであるポリシスケジューラを導入した．ポリシオブジェクトに調整のためのパラメータを置き，これをアプリケーションレベルで調整することにより，アプリケーションレベルでのスレッドスケジューラ間の調整も可能とした．さらに，ポリシスケジューラもアプリケーションレベルで作成するための枠組みを用意することにより，パラメータの調整だけではアプリケーションの要求を満足できない場合にも対応した．

今後の課題としては，以下のものがあげられる．

- 種々のスケジューリングポリシの実装
- 基本関数群の整備
- 種々のアプリケーションを用いた詳細な評価
- 未実装のpthreadインタフェースの実装
- 仮想プロセッサ数が制限されるシステムへの適用
- 実用的なアプリケーションへの応用

参考文献

1) Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tovaniam, A. and Young, M.: Mach: A new kernel foundation for unix development, *Proc. Summer 1986 USENIX Technical Conf.*, pp.93–112 (1986).
2) Cooper, C. and Draves, P.: Cthreads, Technical Report, CMU-CS-88-154, Carnegie Melon University (1988).
3) Mueller, F.: A library implementation of POSIX threads under UNIX, *Proc. Winter 1993 USENIX Technical Conf.*, pp.29–41 (1993).
4) 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).

5) 坂本 力，宮崎輝樹，桑山雅行，最所圭三，福田 晃：並列性と移植性をもつユーザレベルスレッドライブラリPPLの設計および実装，信学論（D-I），Vol.J80-D-I, No.1, pp.42–49 (1997).
6) 佐井範行，最所圭三，福田 晃：ユーザレベルスレッドライブラリPPLの実装：独立したスケジューラの実現，情報処理学会研究報告，96-OS-73, pp.133–138 (1996).
7) ANSI/IEEE Std 1003.1: *Information technology – Portable Operating System Interface (POSIX), Part1: System Application Program Interface (API) [C Language]*, IOS/IEC 9945-1 (1996).
8) 小熊 寿，海江田章裕，森本浩通，田村友彦，鈴木 貢，中山泰一：SMP型計算機を活用する軽量プロセス・ライブラリ，情報処理学会論文誌，Vol.39, No.9, pp.2718–2726 (1998).
9) 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の実現法，信学論（D-I），Vol.J73-D-I, No.12, pp.961–970 (1990).
10) LinuxThreads: http://pauillac.inria.fr/~xleroy/linuxthreads/.
11) Linux: http://www.linux.org/.
12) IEEE: *Threads extension for portable operating systems*, P1003.4a/D6 (1996).

(平成10年12月3日受付)
(平成11年 4 月1日採録)

北口 修一

1997年大阪府立大学工学部電気電子システム工学科卒業．1999年奈良先端科学技術大学院大学情報科学研究科修士課程修了．並列処理一般，システムソフトウェアに興味を持つ．現在，富士通（株）に勤務．

松浦 慎二

1997年大阪教育大学教育学部教養学科卒業．1999年奈良先端科学技術大学院大学情報科学研究科修士課程修了．並列処理一般，システムソフトウェアに興味を持つ．現在，シャープ（株）に勤務．

最所 圭三（正会員）

1959年生．1982年九州大学工学部情報工学科卒業．1984年同大学院工学研究科修士課程修了．同年同大学工学部助手．1991年同大学工学部講師．1993年同大学大型計算機センター助教授．1994年奈良先端科学技術大学院大学情報科学研究科助教授，現在に至る．工学博士．高信頼性システム，並列/分散処理，モバイルシステム，並行処理等の研究に従事．1998年情報処理学会全国大会大会優秀賞受賞．電子情報通信学会会員．

福田 晃（正会員）

1954年生．1977年九州大学工学部情報工学科卒業．1979年同大学院工学研究科修士課程修了．同年NTT研究所入所．1983年九州大学大学院総合理工学研究科助手．1989年同大学助教授．1994年より奈良先端科学技術大学院大学情報科学研究科教授，工学博士．オペレーティング・システム，並列化コンパイラ，計算機アーキテクチャ，並列/分散処理，性能評価等の研究に従事．本学会平成2年度研究賞，平成5年度Best Author賞受賞．著書「並列オペレーティングシステム」（コロナ社），訳書「オペレーティングシステムの概念」（共訳，培風館）．AMC，IEEE Computer Society，電子情報通信学会，日本OR学会各会員．