システムソフトウェアと
オペレーティング・システム 61−14
（１９９３．８．１８）

異なる仮想プロセッサに適応できるスレッド・ライブラリ

宮崎 輝樹† 坂本 力† 最所 圭三‡ 福田 晃†*

† 九州大学工学部情報工学科
‡ 九州大学工学部中央計数施設
* 奈良先端科学技術大学院大学

最近のオペレーティング・システムの多くはスレッド機能を提供する．しかしオペレーティング・システムによってスレッドの定義やインタフェースが異なる．そこでオペレーティングシステムの上に統一的なスレッド・インタフェースを提供するユーザレベル・スレッド・ライブラリを設計した．本ライブラリの特徴として，オペレーティング・システムが提供する“仮想プロセッサ”を統一的に扱い，システムのプロセッサ数に応じた仮想プロセッサを用意することによって，ユーザレベル・スレッドの並列実行を可能にしている．更にライブラリ内の仮想プロセッサ依存の部分をその他の部分と明確に分離することによって，様々なオペレーティング・システムへの移植を容易にする．

本稿ではライブラリの設計および実装について述べる．

Adaptive Thread Library to Variety Virtual–Processor

Teruki Miyazaki† Tikara Sakamoto† Keizo Saisho‡ Akira Fukuda†*

† Department of Computer Science and Communication Engineering,
Faculty of Engineering, Kyushu University
‡ University Computation Center, Faculty of Engineering,
Kyushu University
* Advanced Institute of Science and Technology, Nara
Correspondence: {teruki, sakamoto, fukuda}@csce.kyushu-u.ac.jp,
sai@keisu.kyushu-u.ac.jp

Most of modern operating systems support thredas. However, definitions of a thread and thread interfaces for application programs are different from each other. Thus the authors designed the user–level thread library which provides standard thread interfaces on different operating systems. The characteristic of the library is that user–level threads can be executed in parallel by the following methods.

- Standardized virtual processors provided by an operating system.
- Decision of the number of virtual processors according to the number of real processors.

By separating dependent part and independent part on a virtual processor in the library, portability of the library can be improbed.

This paper describes design and implementation of the library.

1 はじめに

マルチプロセッサ・システムにおいて並列プログラムを記述する場合，並列に実行すべき処理の流れをプロセスを用いて表現する方法が一般的である．並列に実行すべき処理が発生すると，新たなプロセスを生成して実行させ，処理が終了するとプロセスを消滅させる．並列実行の過程では，メッセージ通信によって相互にデータを交換したり，同期をとったりする．このモデルは単純でわかりやすく，また一般性もあるため，現在でも広く使われている．しかし，並列実行の単位が小さく，また並列度が高くなるにつれ，プロセスの生成や消滅，プロセス間での通信の頻度が高くなり，それらの処理にかかるオーバヘッドが無視できなくなる．

そこで，しばしば並列に実行すべき処理の流れを，プロセスの代わりにスレッド (軽量プロセス) を用いて表現する．スレッドとはプロセスからプログラムの実行軌跡のみを取り出したものであり，他の資源の管理はプロセスによって行なわれる．そのため，プロセスよりも生成や消滅といった操作のオーバヘッドが小さい．更に，スレッドはアドレス空間に複数生成することができ，共有しているデータを通して通信や同期などを行なうことにより，プロセスよりも効果的に並列プログラムを実行することができる．特に，協調動作するスレッドはアドレス空間を共有しているので，共有メモリ型マルチプロセッサ・システムにおいて効果を発揮する．

Mach [1] など近年のオペレーティング・システムの多くがこのようなスレッド機能をカーネル内で実現している (以下カーネル・スレッド)．しかしそれらのオペレーティング・システムの間でスレッドの定義やインタフェースがそれぞれ異なる．そのために，あるオペレーティング・システムの提供するスレッドを用いて実現したアプリケーションを，異なるスレッド機能を提供するシステムや，スレッド機能を持たないシステムに移植することは困難である．したがって，これらのオペレーティング・システムの上に何らかの形で，均一なスレッド・インタフェースを提供する必要がある．

本稿では，オペレーティング・システムが提供する様々な種類の仮想プロセッサ (カーネル・スレッド，またはプロセス) の差異を隠蔽し，ユーザやアプリケーションに統一的なインタフェースを提供するユーザレベル・スレッド・ライブラリの構成方法について述べる．本ライブラリは，様々なオペレーティング・システム上で統一的なユーザレベル・スレッド・インタフェースを提供すると同時に，共有メモリ型マルチプロセッサ・システム上で効果的な並列処理を提供する．

2 本スレッド・ライブラリの実現方法

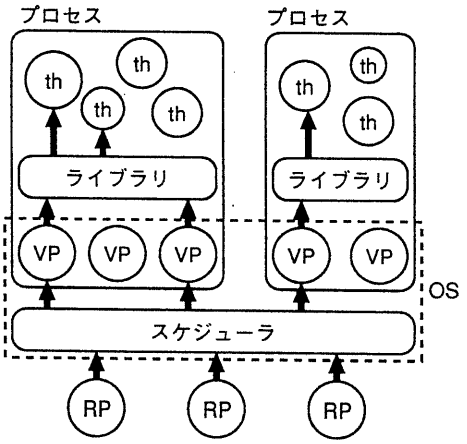
2.1 設計方針

様々なオペレーティング・システム上で標準的なインタフェースを提供するスレッド・ライブラリを設計する．本ライブラリの目標は，以下の2つである．

- 標準スレッド・インタフェースを提供
- 効果的なスレッド実行

1つめの目標を達成するために，各オペレーティング・システムで提供される様々な仮想プロセッサを統一的に扱い，各仮想プロセッサの差異をユーザから隠蔽することにより，ユーザに仮想プロセッサに依存しない標準的なインタフェースを提供する．ここで仮想プロセッサとは，オペレーティング・システムがユーザに提供する仮想的なプロセッサの意味で，そのオペレーティング・システムにおいて処理を実行する単位であるプロセス，またはカーネル・スレッドを指す．

次に2つ目の目標を実現する方法について考えてみる．スレッドの実現方法には，1章で述べたカーネルで制御する方法の他に，ユーザ空間で実現する方法 (以下ユーザレベル・スレッド) がある．ユーザ空間ではコルーチンとしてスレッドを実現する．これらを比較した場合，ユーザレベル・スレッドはカー



th: スレッド
VP: 仮想プロセッサ
RP: 実プロセッサ

図 1 並列処理の様子

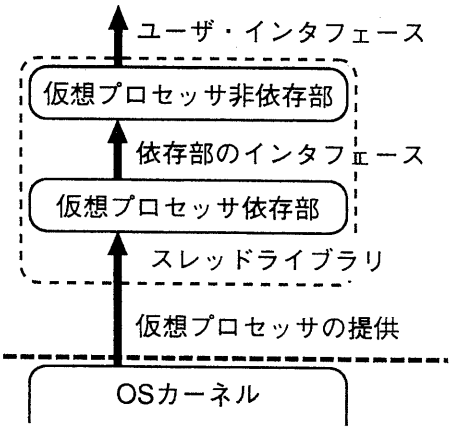


図 2 スレッド・ライブラリの構造

ネル・スレッドよりもスレッド操作のオーバヘッドが小さい．これはユーザレベル・スレッドのスレッド操作が関数呼び出しで行なわれるのに対し，カーネル・スレッドでは全てのスレッド操作にユーザ空間からカーネル空間への空間切替えが伴うためである．特に細粒度・高並列な並列処理を行なう場合，カーネル・スレッドではスレッド操作のオーバヘッドが無視できない．しかしその反面，ユーザレベル・スレッドは直接プロセッサに割り当てることができない．つまりユーザレベル・スレッドを直接並列実行することができない．そこでカーネル・スレッドを実プロセッサの割り当てのエントリとみなし，複数生成したカーネル・スレッドにユーザレベル・スレッドを割り当てることにより，並列処理を実現する (図1参照)．このような方法で実現されたスレッドは，ユーザレベル・スレッドと同じくらいオーバヘッドが小さく，かつ並列実行が可能である．[2]–[6].

我々は，上記の 2 つの目標達成のための手段を統一した．つまり，ライブラリの仮想プロセッサに依存する部分を，非依存の部分から明確に分離し，非依存部に対して統一的な仮想プロセッサ操作のインタフェースを提供する．それにより，各オペレーティング・システムの提供する様々な仮想プロセッサに移植することが容易になる．更に，複数の仮想プロセッサでユーザレベル・スレッドを実行可能とすることにより，カーネル・スレッドが仮想プロセッサである場合，ユーザレベル・スレッドを並列に実行することができる．プロセスが仮想プロセッサであった場合は，本ライブラリのスレッドは通常のユーザレベル・スレッドとして動作する．

2.2 スレッド・ライブラリの構造

本スレッド・ライブラリの構造は図 2の様に，仮想プロセッサに依存する部分と非依存の部分とに分割されている．

仮想プロセッサ依存の部分では仮想プロセッサの生成や消滅，実行の停止および再開といった処理を行ない，非依存部分に対して抽象的な仮想プロセッサ操作のインタフェースを提供する．この部分でオペレーティング・システムごとの仮想プロセッサの違いを吸収し，非依存部の方には抽象的な仮想プロセッサのみを見せる．

非依存部分は多田らによるコルーチン・ライブラリの実現方法[7] を参考にし，setjmp,

th_init()	スレッド・ライブラリの初期化
th_fork(func, arg)	スレッドの生成
th_exit(ret)	スレッドの消滅
th_wait(child)	子スレッドを待機
th_yield()	再スケジュール
th_sched(new)	スレッド切替え

表 1 スレッド操作のインタフェース

vp_alloc()	仮想プロセッサの確保
vp_free()	仮想プロセッサを解放
vp_sleep()	仮想プロセッサの停止
vp_wakeup()	停止している仮想プロセッサの再起動

表 2 依存部分のインタフェース

longjmp を用いてユーザレベル・スレッドを実現した．setjmp，longjmp は C 言語の標準ライブラリとして用意されているので，移植性が良い．また複数の仮想プロセッサによって実行することを前提としているので，レディキューといった共有すべきデータへのアクセスは，相互排他的に行なわれる．

3 スレッドライブラリのインタフェース

スレッド操作に関するユーザインタフェース，および仮想プロセッサ依存部分の提供するインタフェースを表 1，2に列挙した．以下で各関数の詳細を述べる．

3.1 ユーザインタフェース

- int th_init()

  スレッド・ライブラリの初期化を行なう．各データ構造を確保し，vp_alloc を呼び出して仮想プロセッサを生成する．

- thread_t th_fork(func, arg)
  void (*func)();
  any_t arg;

  関数 func を実行する新しいスレッドを生成し，レディキューに接続する．arg は func への引数である．この関数では，呼び出されるとまずスタックとスレッドの状態を保存するコントロール・ブロックを確保する．スタックサイズはシステムで固定とする．スタックは底にスレッドのスタートルーチンを実行するための関数フレームを生成し，コントロール・ブロック内のコンテキストを初期化する．スタートルーチンとは，全てのスレッドが最初に実行する関数で，そこから func および th_exit が呼び出される．

- void th_exit(ret)
  int ret;

  各スレッドは明示的にスレッド消滅の関数 th_exit を呼び出すことにより終了する．また明示的に呼び出さなくても，スレッドとして定義された関数の終了時に自動的に th_exit を呼び出す．ret はスレッドの終了値である．また全てのスレッドが終了した場合，vp_free にて仮想プロセッサを解放し，アプリケーションの実行を終了する．

- int th_wait(child)
  thread_t child;

  子スレッド child の終了を待機する．th_wait は child の終了値を返す．

- void th_yield()

  スレッド切替えを行なう．スレッドの実行を中断する時に用いる．

- void th_sched()

  スレッドのスケジューリングとコンテキスト切替えを行なう．この関数はユーザには提供しない．スケジューリングは現在ラウンドロビンで行なっており，他の方式はサポートしていない．コンテキスト切替えは setjmp，longjmp で行なっている．

3.2 依存部分のインタフェース

- int vp_alloc()

  スレッド・ライブラリの初期化の時に仮想プロセッサを生成する．生成する仮想プロセッサの個数はシステムの実プロセッサ数に固定する．これは実プロセッサ数以上の仮想プロセッサを生成しても，実プロセッサ数を越えた並列性は出ないためである．またオペレーティング・システムによってプロセスに割り当てることのできる実プロセッサ数の上限がある場合，その数だけの仮想プロセッサを生成する．

- int vp_free()

  アプリケーションの終了時にvp_allocで生成した仮想プロセッサを消滅する．

- int vp_sleep()

  実行すべきユーザレベル・スレッドがない場合に，一定時間仮想プロセッサを停止させる．これは仮想プロセッサがスケジューラ内で無駄なループを繰り返しプロセッサを浪費することを防ぐためである．設定した時間の経過後または他の仮想プロセッサがvp_wakeupを実行した場合に，実行を再開する．

- int vp_wakeup()

  停止している仮想プロセッサが存在する場合，それらの実行を再開させる．空であったレディキューにスレッドを接続した場合に呼び出す．

シングルプロセッサ・システムの場合，アプリケーション起動時のプロセスのみでユーザレベル・スレッドの実行を行なうため，これらの関数は何も行なわない．

4 他のシステムとの比較

4.1 ユーザレベル・スレッドとの比較

ユーザレベル・スレッドと本方式とを比較する．シングルプロセッサ・システム上で実現した場合，本方式は単なるユーザレベル・スレッドとして動作するため，2つの方式の間にほとんど差はない．しかし共有メモリ型マルチプロセッサ・システム上のオペレーティング・システムがカーネル・スレッドを提供している場合，本方式のユーザレベル・スレッドは並列に実行することが可能である．ただし，本ライブラリは複数の仮想プロセッサ上で動作させることを前提にしている．そのため，レディキューやスレッドのコントロール・ブロックといった共有するべきデータにアクセスする場合，排他制御をとる必要がある．したがって，逐次的に実行するために排他制御をとる必要のないユーザレベル・スレッドと比べると，本方式におけるスレッド操作のオーバヘッドは少し大きい．

4.2 カーネル・スレッドとの比較

カーネル・スレッドと本方式とを比較する．カーネル・スレッドはスレッド操作をカーネル内で行なうために，全てのスレッド操作にユーザ空間からカーネル空間への空間切替えが生じる．また悪意やバグのあるプログラムから他のスレッドやプロセスなどを保護するため，引数のチェックを厳重に行なう必要がある．それに対し，本方式におけるスレッド操作はライブラリに対する関数呼び出しによって行なわれる．またオペレーティング・システムによって保護されているので，引数のチェックはほとんど必要ない．したがってスレッド操作のオーバヘッドは本方式の方が小さい．また本方式ではアプリケーションに合わせてスレッド・システムを変更することが容易に行なえる．

それに対しカーネル・スレッドの利点は，スケジューラを起こすイベントを捕らえることが容易なことにある．スケジューラを起こすイベントとしては，タイマ割り込みや，スレッド上のプログラムが発行したシステムコールのブロック，アンブロック，他のスレッドからのシグナルなどが挙げられる．これらのイベントを捕らえスケジューラを起動することにより，効果的にスレッドを制御することができる．

4.3 その他のスレッドとの比較

新城らは文献 [5] [6] で，仮想プロセッサ方式を提案している．仮想プロセッサ方式では，実プロセッサを割り当てるためのエントリとして仮想プロセッサを定義し，その上でマイクロプロセスをユーザレベルで実行することにより，並列実行を実現している．仮想プロセッサとカーネル・スレッドとの違いは，仮想プロセッサはその上でユーザレベル・スレッドを実行することを前提に設計されていることにある．またマイクロプロセス・ライブラリによって応用固有のスケジューリングを提供する．

仮想プロセッサ方式は，オペレーティング・システムにおいて仮想プロセッサを提供し，その上でユーザレベルのスレッドを実行する点で，本方式とほぼ同じである．ただ，仮想プロセッサに対する考え方が異なる．我々は既存の仮想プロセッサ上でユーザレベル・スレッドを実行しているのに対し，新城らはユーザレベル・スレッドをより効果的に実行するための，仮想プロセッサのインタフェースを定義し，そのような仮想プロセッサを提供するカーネルを実現している．

5 実現

本ライブラリを以下のマシン上で実現した．

1. Sun IPC, SunOS 4.1.2, Sparc × 1
2. Sequent S-2000, DYNIX 3.1.1, i486 × 20

Sun IPC はシングルプロセッサ・システムであるため，仮想プロセッサ (= プロセス) の最大数は 1 である．したがって，仮想プロセッサの制御を行なう vp_alloc，vp_free，vp_sleep，vp_wakeup は機能しない．つまり本ライブラリは通常のユーザレベル・スレッド・ライブラリとして動作する．

Sequent はマルチプロセッサ・システムであるが，オペレーティング・システムの DYNIX はカーネル・スレッドを提供していない．その代わりに，データを共有することができる複数のプロセス群を生成して，それらを並列実行する機能を提供している．Sequent 上に実現した本ライブラリでは，このプロセス群を仮想プロセッサとみなす．仮想プロセッサの生成 vp_alloc は，共有プロセス群を生成するシステムコール m_fork [8] によって実現した．m_fork は，事前に設定した個数の子プロセス群を生成し，全ての子プロセスに引数で指定した関数を実行させる．この時プログラム中で明示的に指定されたデータは全ての子プロセスで共有される．また動的に共有領域を割り当てることもできる．vp_free では，m_fork によって生成した全ての子プロセスを消滅させるシステムコール m_kill_procs を呼んでいる．また vp_sleep，vp_wakeup は alarm と kill を用いて実現した．ユーザレベル・スレッドのスタック領域の確保として，動的に共有メモリの割り当て，解放を行なう shmalloc および shfree を用いて行なっている．

6 今後の課題

本ライブラリを，現在シングルプロセッサ・システムでありプロセスを仮想プロセッサとする Sun と，マルチプロセッサ・システムでメモリの共有と並列実行が可能なプロセスを提供する Sequent の 2 機種上で実現した．今後本スレッド・システムの移植性を確認するために，より多くのオペレーティング・システム上に移植を行なう必要がある．現在マルチプロセッサ・システム上でカーネル・スレッドを提供するオペレーティング・システムへ移植することを予定している．

またユーザレベル・スレッドのスケジューリング方式は現在ラウンドロビンに固定されており，アプリケーションによってスケジューリングを変更することができない．しかし，固有のスケジューリングを実現することによって著しく性能が向上するアプリケーションが存在することがわかっている．したがってユーザにスケジューリング方式を選択させる，またはスケジューリング・ポリシーを決定させる何らかのインタフェースを設ける必要があると思われる．

7 おわりに

統一的なスレッド・インタフェースを様々なオペレーティング・システム上に提供するために，移植性を考慮したスレッド・ライブラリの構成方法について述べた．本ライブラリでは，オペレーティング・システムの提供する，処理を実行する単位であるプロセス，またはカーネル・スレッドを仮想プロセッサとみなし，スレッド・ライブラリ内の仮想プロセッサに依存する部分を他の部分から明確に分離することによって，様々なオペレーティング・システムへの移植を容易にしている．また仮想プロセッサがカーネル・スレッドである場合，複数の仮想プロセッサにユーザレベル・スレッドを割り当てることにより，並列実行を可能とする．以上のような設計方針の元でスレッド・ライブラリの実装をSunおよびSequent上にて行なった．特にSequentでは，並列実行可能なプロセッサ群の生成をオペレーティング・システムが提供している．このプロセッサ群の間でスレッドのデータ構造を共有することにより，ユーザレベル・スレッドの並列実行を実現した．今後本ライブラリを様々なオペレーティング・システムへ移植することを予定している．

参考文献

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young: Mach : A New Kernel Foundation for UNIX Development, Proceedings of the Summer 1986 USENIX Technical Conference,pp.93-112(1986)

[2] B. Marsh, M. Scott, T. LeBlanc and E. Markatos: First–Class User–Level Threads, Proceedings of the 13th ACM Symposium on Operating Systems Principles, October(1991).

[3] T. Anderson, B. Bershad, E. Lazowska and H. Levy: Scheduler Activations: Effective Kernel Support for the User–Level Management of Parallelism, Proceedings of the 13th ACM Symposium on Operating Systems Principles, October(1991).

[4] E. Cooper and R. Draves: C Threads, Technical Report CMU–CS–88–154 , School of Computer Science, Carnegie–Mellon University(1988).

[5] 新城 靖, 清木 康: 並列プログラムを対象とした軽量プロセスの実現方式, 情報処理学会論文誌, Vol. 33, No. 1,pp.64-73(1992).

[6] 新城 靖, 清木 康: 仮想プロセッサを支援するオペレーティング・システム・カーネルの構成法, 情報処理学会論文誌, Vol. 34, No. 3, pp. 478-488(1993).

[7] 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実現法, 電子情報通信学会論文誌 D-I, vol.J73-D-I, No.12, pp.961-970(1990).

[8] SEQUENT COMPUTER SYSTEMS: SEQUENT GUIDE TO PARALLEL PROGRAMMING, (1989)