システムソフトウェアと
オペレーティング・システム 63－7
（１９９４． ３． ２２）

# 様々な仮想プロセッサに適応できるスレッドライブラリの実現

宮崎 輝樹[†] 桑山 雅行[†] 最所 圭三[‡] 福田 晃[†*]

† 九州大学工学部情報工学科
‡ 九州大学大型計算機センター
* 奈良先端科学技術大学院大学

我々は，OS の提供する様々な仮想プロセッサ・モデルに対応できるスレッドライブラリ PPL(Parallel Pthread Libaray) を設計した．PPL の設計時の要件として (1) 並列性，および (2) 移植性を挙げた．(1) を満たすために，OS の提供する仮想プロセッサを複数生成し，その上で並列実行可能なマルチルーチンとしてユーザレベル・スレッドを実現した．また (2) を満たすために，仮想プロセッサやシステムに依存する部分と依存しない部分とを明確に分離し，更にその間のインタフェースを統一した．

本論文では，PPL の設計と実装について述べる．実装対象の仮想プロセッサとして，UNIX プロセス，Mach thread，共有プロセスを選んだ．また，他のスレッドライブラリとの性能比較の結果についても述べる．

# Implementation of a Thread Library with Adaptability in Virtual Processor Models

Teruki Miyazaki[†] Masayuki Kuwayama[†] Keizo Saisho[‡] Akira Fukuda[†*]

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

We implement a thread library, PPL(Parallel Pthread Library), which can be adapted to various virtual processor models. PPL is intended to be with parallelism and portability. To satisfy parallelism, PPL employs multiroutine approach which allows multiple threads to be executed in parallel. To satisfy portability, we partition PPL into a part which depends on virtual processor and the other. We define interface between them.

In this paper, we describe about design and implementation of PPL. We implement PPL on three virtual processor models; UNIX-like process, Mach thread, and shared process. Through the implementation, we compare the performance of PPL with that of existing thread libraries.

## 1 はじめに

近年，スレッド (軽量プロセス) の概念が注目を集めている．スレッドとは，従来のプロセスからプログラムの実行軌跡のみを分離したものであり，従来のプロセスと比べて生成や消滅，コンテキスト切替え等のオーバヘッドが小さい．また，プロセス上のスレッドはアドレス空間を共有しているため，共有データを通じて，スレッド間の通信や同期が高速に，かつ容易に実現できる．これらの特徴のため，従来のプロセスを用いた並行/並列処理に比べ効率的な並行/並列処理が行なえる．

スレッドは実現方式により，カーネル・スレッド，ユーザレベル・スレッドに大きく分類することができるが，カーネル・スレッドのスレッド操作のオーバヘッドはユーザレベル・スレッドと比べ非常に大きいため，ユーザレベル・スレッドの方が有効であると一般的に言われている．

しかし，現在実現されているユーザレベル・スレッドのほとんどがコルーチンとして実現されているため，スレッドは逐次的に実行され，並列実行が行なわれない．また，各々のスレッドシステムが独自のスレッドインタフェースを提供しているため，スレッドを用いたアプリケーションの移植が困難である．

そこで我々は，(1) システムに応じた並列実行が可能であること，および (2) 多くの OS 上に共通のスレッド環境を提供するために移植性が良いこと，の 2 つを要件としてスレッドライブラリ PPL(Parallel Pthread Library) を設計，実装した [1]．PPL では，(1) を満たすために，OS の提供する複数の仮想プロセッサ上で並列実行可能なマルチルーチンとしてユーザレベル・スレッドを実現した．また (2) を満たすために，OS の提供する仮想プロセッサに依存する部分と依存しない部分とを明確に分離し，更にそれらの間のインタフェースを統一した．

本論文では，PPL の設計方針とそれを満たす構成について述べる．また，実装方法について述べ，実装上の問題点を挙げる．更に，他のスレッドライブラリと性能および移植性に関する比較を行ない，考察を述べる．

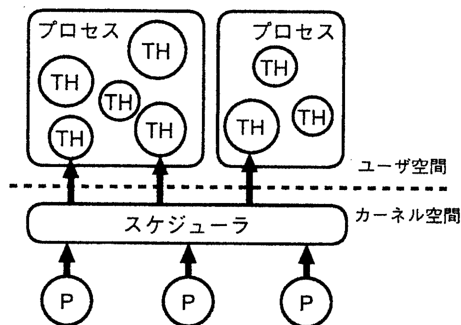


図 1 カーネル・スレッド

## 2 PPL の設計

### 2.1 設計要件

現在，多くのスレッドシステムが実現されている．それらは，カーネル・スレッドとユーザレベル・スレッドに大きく分類することができる．カーネル・スレッドは，OS のカーネルによってユーザに提供されるスレッドで，Mach や Chorus といったマルチプロセッサを対象とした多くの OS で実現されている (図 1)．また，ユーザレベル・スレッドは，ユーザ空間で実現したスレッドで，多くの場合 SunOS の LWP(Light Weight Process) の様に，ライブラリとして提供される (図 2)．

カーネル・スレッドの操作はカーネル内で行なわれるため，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．また，不正な操作から他の資源を保護するために，引数の厳重なチェックが必要である．これらの理由により，カーネル・スレッド操作のオーバヘッドは，スレッド操作がユーザ空間で行なわれるユーザレベル・スレッドと比べ，非常に大きい．そのため，カーネル・スレッドよりもユーザレベル・スレッドの方が効果的なスレッド操作が行なえると一般的に言われている．

しかし，現存のユーザレベル・スレッドライブラリの問題点として，以下の 2 つが挙げ

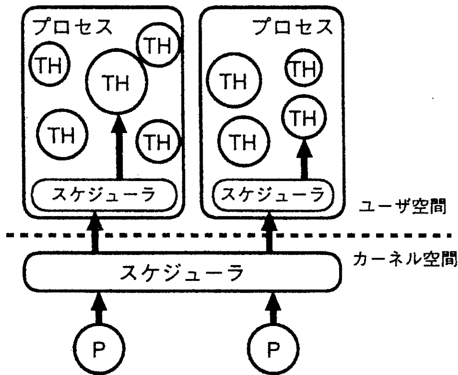


図 2 コルーチンによるスレッド

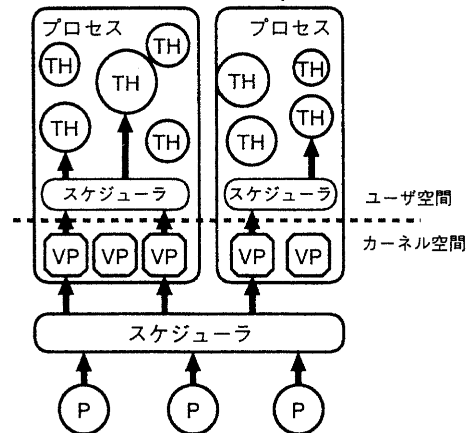


図 3 マルチルーチンによるスレッド

られる.

- 現存のスレッドライブラリのほとんどが，従来のプロセス上のコルーチンとしてスレッドを実現している．コルーチンとして実現されたスレッドは逐次的に実行されるため，マルチプロセッサ・システムのプロセッサ資源を有効に利用することができない．

- ほとんどのスレッドライブラリが独自のスレッドインタフェースを提供している．そのため，移植性の良いスレッドを用いたアプリケーションの実現が困難である．

以上のようなユーザレベル・スレッドライブラリの問題点を基に，PPL の設計の要件として以下の 2 つを挙げた．

**並列性** システムに応じた並列実行が可能なユーザレベル・スレッドを実現する．

**移植性** より多くの OS 上に共通のスレッド環境を実現するために，移植性の良いスレッドライブラリを実現する．

### 2.2 設計方針

PPL の要件である並列性，および移植性を実現する設計方針について述べる．

まず，1 つ目の要件である並列性を満たすために，複数の仮想プロセッサ上で並列に実行できるマルチルーチンとして，ユーザレベル・スレッドを実現する．ここで仮想プロセッサとは，OS が実プロセッサを多重化しユーザに仮想的なプロセッサとして提供するプロセスやカーネル・スレッドの総称である．また，マルチルーチンとは，1 つの仮想プロセッサ上で逐次的に実行される手続きの集まりであるコルーチンを，複数のプロセッサ上で並列実行可能なように一般化したものである．それにより，カーネル・スレッドの並列性と，ユーザスレッドの軽量性を兼ね備えたスレッドシステムを実現することができる (図 3)．

次に，2 つ目の要件である移植性の実現について述べる．様々な仮想プロセッサ・モデルに対応させる場合，移植時の問題となるのは，各 OS 毎に仮想プロセッサの操作のインタフェースが異なることである．そこで，本ライブラリでは，仮想プロセッサに依存する部分と依存しない部分とを明確に分離し，その間のインタフェースを統一した．

PPL の構成を図 4に示す．仮想プロセッサに依存する部分を VP(Virtual Processor) 依存部，依存しない部分を VP 非依存部と呼ぶ．また，VP 依存部から VP 非依存部に対して提供する統一的なインタフェースを VP インタフェースと呼ぶ．以下に各部分の役割について述べる．

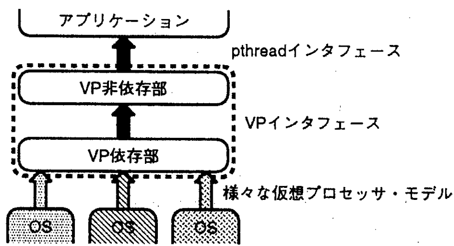


図 4 PPL の構成

### 2.2.1 VP 依存部

VP 依存部とは，システムの仮想プロセッサに依存する部分である．VP 依存部では，OS の提供する仮想プロセッサの違いを吸収する．また，VP 非依存部に対して統一的な VP インタフェースを提供する．異なる仮想プロセッサを提供するシステム上に PPL を移植する場合，VP 依存部を移植するシステムの仮想プロセッサに合わせて記述し，VP インタフェースを提供することにより，VP 非依存部をほとんど変更することなく行なえる．

仮想プロセッサ操作に関する VP インタフェースとしては，以下の 4 つを定義した．

**vp_create** 新たに仮想プロセッサを生成する．生成された仮想プロセッサは，実行可能状態として VP レディキュー (ready queue) に接続される．

**vp_terminate** 指定した仮想プロセッサを消滅させる．

**vp_suspend** 本関数を呼び出した仮想プロセッサを一時停止させる．停止された仮想プロセッサは，実行可能状態として VP レディキューに接続される．ライブラリのスケジューラで次に実行するスレッドがない場合に呼び出す．

**vp_resume** 停止している仮想プロセッサの実行を再開させる．実行を再開させる仮想プロセッサは VP レディキューから選ぶ．また，VP レディキューが空の場合，何も行なわない．新しいスレッドを生成した場合や，停止していたスレッドの実行を再開させる場合に呼び出される．

### 2.2.2 VP 非依存部

VP 非依存部とは，システムの仮想プロセッサに依存しない部分である．VP 非依存部では，VP 依存部の提供する仮想プロセッサ上で並列動作するマルチルーチンとしてユーザレベル・スレッドを実現する．VP 非依存部は移植性が良くなければならない．そのためシステムに依存する部分をできるだけ少なくし，全てを C 言語で記述する．

ユーザに提供するスレッドインタフェースとしては，pthread インタフェースを採用した．pthread インタフェースとは，IEEE の策定している UNIX の統一的な規格 POSIX で規定された標準的なスレッドインタフェースである．現在，pthread インタフェースはまだドラフト段階であり，pthread を提供する UNIX もほとんど実現されていない．しかし，今後発表される多くの UNIX が pthread インタフェースを提供するものと思われる．また，pthread インタフェースを提供するスレッドライブラリが既に幾つか実現されている．そのため，アプリケーションの移植性を考え，PPL では pthread インタフェースを採用した．

## 3 VP 依存部の実現

### 3.1 対象とした仮想プロセッサ・モデル

今回 PPL の実装対象の仮想プロセッサ・モデルとして，UNIX プロセス，Mach thread，DYNIX の共有プロセスを選んだ．以下にそれぞれの特徴と，実装したシステムについて述べる．

**UNIX プロセス** 従来の UNIX プロセスを仮想プロセッサとみなす．この場合，仮想プロセッサは 1 つであり，VP 依存部は何も行なわない．

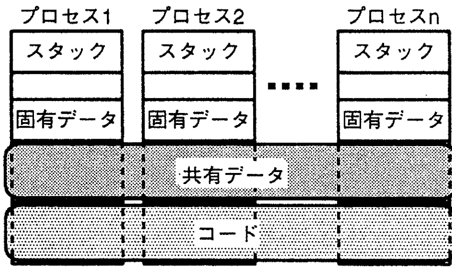


図 5 共有プロセス

**Mach thread** カーネル・スレッドである Mach thread を仮想プロセッサとみなす．

**共有プロセス** Sequent 社の S-2000 は複数のプロセッサをバスで結合した共有メモリ型マルチプロセッサであり，S-2000 上の OS・DYNIX では，並列プログラミング支援機能を提供している．その 1 つとして，コードおよび一部のデータを共有した複数のプロセスを生成し，プロセッサに割り当て，指定した関数を並列に実行させる機能を備えている (以下共有プロセスと呼ぶ)．共有プロセスの様子を図 5に示す．この共有プロセスを仮想プロセッサとみなす．

### 3.2 Mach thread 依存部の実現

**vp_create** Mach thread の生成プリミティブである thread_create を用いて実現した．thread_create では，停止状態の Mach thread が生成されるのみで，スレッドの起動は行なわれない．今回の実装では，本関数はライブラリ初期化時の一度だけ呼び出すものとし，一度の呼び出しである決まった個数の Mach thread を生成するようにした．

**vp_terminate** Mach thread の終了プリミティブである thread_terminate を用いて実現した．今回の実装では，本関数はライブラリの終了時の一度だけ呼びだし，一度の呼び出しで全ての Mach thread を終了させる．

**vp_suspend** Mach thread を停止させるプリミティブである thread_suspend を用いて実現した．

**vp_resume** 停止中の Mach thread の実行を再開させるプリミティブである thread_resume を用いて実現した．

### 3.3 共有プロセス依存部の実現

**vp_create** 共有プロセスを生成する m_fork を用いて実現する．m_fork では指定した関数から実行を開始する共有プロセスを，指定した個数だけ同時に生成し，プロセッサに固定的に割り付ける．従って，DYNIX 上の PPL では，本関数を初期化時の一度だけ呼び出す．生成された共有プロセスはすぐに vp_suspend を呼び出して停止状態にされる．

**vp_terminate** 共有プロセスを消滅させる m_kill_procs を用いて実現する．m_kill_procs を実行すると，m_fork で生成した全ての共有プロセスを終了させる．そのため，DYNIX 上の PPL では，本関数をライブラリ終了時の一度だけ呼び出す．

**vp_suspend** DYNIX では特定の共有プロセスのみを停止させる機能を用意していない．従って，PPL ではロックされているロック変数に対しロック獲得の関数を実行し，ビジーウェイトさせることにより，疑似的に停止状態にしている．

**vp_resume** vp_suspend でビジーウェイトしているロック変数をアンロックすることにより，指定した共有プロセスを停止状態から解放する．

# 4 VP 非依存部の実現

VP 非依存部では，移植性の良いマルチルーチンを実現しなければならない．ここでは移植時に問題となるシステムに依存する部分の実現方法，および VP 非依存部の実現時の問題点に関して述べる．

## 4.1 コンテキスト切替え

通常コンテキスト切替えは，プロセッサのレジスタ情報を保存·復元するためにシステムに大きく依存し，移植時の問題となる．PPL では，移植性の良いコンテキスト切替えの方法として，setjmp · longjmp を用いたコンテキスト切替えを行なう [2].

## 4.2 スタック

各マルチルーチンには，独立したスタックを割り当てる必要がある．スタックとしては，(1) データ領域，(2) ヒープ領域，または (3) 共有メモリ領域に確保する 3 種類の方法が考えられる．アプリケーション内でスレッドを実現する場合など，あらかじめ生成するスレッドの数や，必要なスタックの大きさなどがわかっている場合 (1) で静的に用意する方法も使えるが，スレッドライブラリではあらかじめアプリケーションで必要なスレッドの数やスタックの大きさなどはわからないので，動的に確保や解放が行なえる (2) または (3) で実現する．(2) と (3) を比べた場合，(2) ではスタックの溢れ (オーバフロー) を自動的に検出することができないのに対し，(3) では確保した領域の下限をスタックが越えた場合，不正領域アクセスのシグナルが起きるので，オーバフローを検出することができる．ただし (3) の実装時の問題点として，共有メモリ機構の移植性が挙げられるが，現在のほとんどの UNIX-like な OS で提供されているので，PPL では (3) の方法を採用した．

## 4.3 スタックポインタの設定

生成したスレッドをはじめて実行する場合，スタックポインタ (SP) がそのスレッド用に確保したスタックを指すように SP レジスタを変更する必要がある．通常その方法として直接 SP レジスタを書き換える方法が取られるが，その場合 SP を書き換える手続きをアセンブラにて記述する必要があり，移植時の問題となる．そこで PPL では，setjmp にてコンテキストを保存する領域である jmp_buf 構造体の SP を格納する箇所に新しい SP の値を格納し，longjmp を行なう方法を取った．

## 4.4 排他制御

各マルチルーチンは並列に実行される可能性があるので，マルチルーチン管理に関するデータ構造へのアクセスは，クリティカルセクションとして排他制御を行なう必要がある．その際，排他制御を取る単位が問題となる．排他制御を取る単位として (1) 各構造体毎，または (2) ライブラリ全体の 2 種類を考えた．(1) の方法では，クリティカルセクションとされる部分が最小限に抑えられ，また異なるデータ構造への並列アクセスが期待されるが，排他制御に必要なロック変数が多くなり，更にデッドロックに注意する必要があるため制御が複雑になる．そこで PPL では，構造の簡単な (2) の方法を採用している．

## 4.5 仮想プロセッサ生成のアルゴリズム

VP インタフェースを用いて仮想プロセッサを生成する個数や，生成するタイミングについて幾つかの選択肢が考えられる．まず生成する仮想プロセッサの個数であるが，システムの物理プロセッサ数以上の仮想プロセッサを生成しても，カーネルによって物理プロセッサ上で多重化される．従って PPL では，生成する仮想プロセッサ数をシステムの物理プロセッサ数と同じとする．また生成するタイミングとして，あらかじめ静的に用意しておく方法と，必要となった場合に動的に生成·消滅を行なう方法とが考えられるが，通常仮

想プロセッサの生成・消滅はカーネル内で行なわれるために，スレッドの生成・消滅と比べ，オーバヘッドが非常に大きい．またPPLでは，生成する仮想プロセッサの個数をはシステムの物理プロセッサ数に固定している．そこでPPLでは，ライブラリの初期化時に静的に全ての仮想プロセッサを生成し，ライブラリ(プログラム)の終了時にそれらの仮想プロセッサを消滅させるものとする．

# 5 PPLの評価

## 5.1 性能評価環境

スレッド操作の性能を他のスレッドライブラリと比較する．比較の対象としてPTL, FPT, Cthreadsを選んだ．それぞれのスレッドライブラリについて説明する．

**PTL[3][4]** PTLとは，大阪大学の安部らによって開発されたpthreadライブラリで，対象をBSD UNIXに限定することにより移植性の良いスレッドライブラリを実現している．また，ユーザレベルでのスレッドの自動拡張や，ノンブロッキングI/O，スレッドの実行履歴を取る機能なども実現している．スレッドの実現方式はコルーチン方式である．

**FPT[5]** Frolida大学のMullerらによって開発されたSunOS専用のpthreadライブラリで，対象をSunOSに限定することにより高速なスレッド操作を実現している．スレッドの実現方式はコルーチン方式である．

**Cthreads[6]** Machにおいて，ユーザのMach thread(カーネル・スレッド)の利用を支援するために用意されたC言語のライブラリである．従ってスレッドの実現方式は，実際にはカーネル方式である．

表1の比較環境にて，スレッド操作の性能として，コンテキスト切替えとスレッド生成の実行時間を測定した．Sequent S-2000上での測定は，今回は行なわなかった．

表1 比較環境

| | SunIPX | luna88k |
|---|---|---|
| OS | SunOS 4.1.3 | Mach 2.5 |
| CPU | Sparc × 1 | MC88100 × 4 |
| 比較対象 | PTL, FPT | PTL, Cthreads |

## 5.2 性能評価

測定の結果を表2に示す．

FPTおよびCthreadsのコンテキスト切替えと比べ，PPLおよびPTLでは3〜7倍遅い．これは，FPTおよびCthreadsではアセンブラによってコンテキスト切替えを記述しているのに対し，PPLおよびPTLでは移植性を考慮してCの標準関数setjmp・longjmpを用いてコンテキスト切替えを行なっているためである．

また，特にluna88k上のPTLのコンテキスト切替えが遅いのは，移植性を保ちつつlongjmpのSPチェックを回避するために，コンテキスト切替えの度にシグナルを起こして，一度シグナルスタックへSPを設定しているためである．PPLでは，vp_createで生成したMach threadがシグナルを受け取れない問題に対処するために，longjmpを書き換えてSPチェックを外しているので，シグナルを起こす必要がなくPTLと比べてコンテキスト切替えが早い．

Cthreadsのスレッド生成が遅いのは，スレッドを生成する度にMach threadを生成しているためである．それに対しPPLでは，動的な仮想プロセッサの生成・消滅は行なわず，ライブラリの初期化時と終了時に静的に行なっているため，スレッドの生成は他のコルーチン方式のスレッドライブラリ(PTL, FPT)と同等である．

表 2 スレッド操作の比較

| | Sun IPX($\mu sec$) | | | luna88k($\mu sec$) | | |
|---|---|---|---|---|---|---|
| | PPL | PTL | FPT | PPL | PTL | Cthreads |
| コンテキスト切替え | 250 | 180 | 46 | 180 | 350 | 50 |
| スレッド生成 | 2100 | 2300 | 1700 | 3000 | 2600 | 6500 |

### 5.3 移植性に関する考察

UNIX のプロセスを仮想プロセッサとした場合，仮想プロセッサは1つでありVP依存部は何も行なわない．そのため，ほとんど変更なく他の UNIX プロセスへ移植することができる．

現在のVPインタフェースは，仮想プロセッサ・モデルとしてカーネル・スレッドを参考に定義している．そのため，仮想プロセッサとしてカーネル・スレッドを提供するシステムのVP依存部は，容易に行なえると思われる．

ただし，共有プロセスのように仮想プロセッサのモデルが異なった場合，VP インタフェースに適合させることが困難である．従って，今後VPインタフェースの定義について再考する必要があるものと思われる．

## 6 おわりに

並列性と移植性を要件としたスレッドライブラリ PPL の設計と実装について述べた．

他のスレッドライブラリと比較した場合，スレッド操作の性能は遜色無いことがわかった．今後，PPL の並列実行の有効性を確認するために，様々なタイプの並列性を持ったアプリケーションに関して，コルーチンによる他のスレッドライブラリと比較評価を行なう必要がある．

また，移植性に関して考察するために，異なる仮想プロセッサを提供する他のシステムへの移植を行なう必要がある．

更に，より多くの仮想プロセッサ・モデルについて考慮し，VPインタフェースの定義を再考する必要があるものと思われる．

## 参考文献

[1] 宮崎 輝樹, 坂本 力, 最所 圭三, 福田 晃: “異なる仮想プロセッサに対応できるスレッドライブラリ”, 情報処理学会 OS 研究会, 61-14, Aug. 1993.

[2] 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, vol.J73-D-I, No.12, pp.961-970(1990).

[3] 安部 広多, 松浦 敏雄: “移植性に優れた軽量プロセスライブラリの実現方”, Proceedings of the 21st jus UNIX symposium, pp.78-89, Jul. 1993.

[4] 安部 広多, 松浦 敏雄, 谷口 健一: “BSD UNIX の下でのポータブルマルチスレッドライブラリ PTL の実現”, 情報処理学会 OS 研究会, 62-10, Jan. 1994.

[5] F. Mueller: “A Library Implementation of POSIX Threads under UNIX”, Winter USENIX, 1993.

[6] E. Cooper and R. Draves: “C Threads”, Technical Report CMU-CS-88-154 , School of Computer Science, Carnegie-Mellon University, Jun. 1988.