Vol. 36 No. 2 情報処理学会論文誌 Feb. 1995

BSD UNIX 上での移植性に優れた軽量プロセス機構の実現

安倍広多[†,*] 松浦敏雄[†] 谷口健一[†]

近年，一つのプロセスの中に，概念的に小さなプロセス（軽量プロセス，スレッドとも呼ばれる）を複数走らせ，それぞれに独立した処理をさせることができる機構が注目されている．UNIX 上でも軽量プロセスを実現するいくつかのライブラリが存在しているが，それらは特定のアーキテクチャに依存しており，移植性が低いという問題点があった．そこで本研究では BSD UNIX 上での移植性の良い軽量プロセス機構の実現法を検討し，実際にそれを実現するライブラリを作成した．ライブラリの実現に当たっては，アーキテクチャに依存せずにスタックポインタを設定する方法，スレッドが用いる各スタックを自動的に拡張する方法，等が問題となるが，それぞれの対策を工夫し解決した．このライブラリは特定のアーキテクチャに依存しないため，軽量プロセスを利用したプログラムも様々な計算機上で稼働させることが可能となった．作成したライブラリが，SunOS 4, Ultrix 4, DEC OSF/1, NEWS-OS 4, BSD/386 等，数多くのアーキテクチャ上で実際に動作すること，および，特定のアーキテクチャに依存した他の軽量プロセス機構と比較しても遜色ない速度で動作することを確認した．

An Implementation of Portable Lightweight Process Mechanism under BSD UNIX

Kota Abe,[†,*] Toshio Matsuura[†] and Kenichi Taniguchi[†]

In many operating systems, lightweight process (also known as thread) mechanism has been implemented. Multiple lightweight processes can exist in one (heavy weight) process and each lightweight process performs individual job. Currently, several thread packages are avalable under UNIX, but these packages have low portability and work on particular UNIX. Therefore portable thread package is widely desired. We have developed a portable library for a lightweight process mechanism on BSD UNIX. This library is not dependent on particular architecture. To implement such a library, following problems should be solved: How to set a CPU's stack pointer appropriately without architecture dependency. How to expand thread's stack automatically, etc. In this paper, we describe their solutions. We have confirmed that this library works on most BSD-based UNIX and its performance is almost equal to other architecture-dependent lightweight process implementations.

1. まえがき

計算機ネットワークの普及に伴い，サーバクライアントモデルに基づくプログラムが広く用いられるようになってきた．このようなプログラムに限らず，一般に複数の作業を協調しながら並行して動作させる必要があるプログラムは少なくない．UNIX の下でこのような処理を行わせるには，それぞれの作業を別々のプロセスに割り当てて実行させるのが一般的であるが，通常，UNIX のプロセスの生成や切替えには相当の時間を要し，このような作業を高速に実行させることは難しい．そこで，一つのプロセスの中に，それぞれが独立した処理を実行できるような，より小さな軽いプロセス（**軽量プロセス**あるいは**スレッド**等と呼ばれる）を複数生成する機構が利用されるようになってきた．各軽量プロセスは，一つのプロセス（軽量プロセスに対して重量プロセスと呼ぶ）内でアドレス空間やその他の大域的なデータを共有しているが，CPU のレジスタ，局所変数，スタックは互いに独立している．重量プロセスに比べて生成，消滅，コンテキストスイッチのオーバヘッドが少ない等の特徴がある．

軽量プロセス機構をカーネルレベルで実現している OS もあるが[5),8)]，この方法ではシステムコールにかかるオーバヘッドのため，速度の点からはユーザレベ

† 大阪大学基礎工学部情報工学科
Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University
* 現在，日本電信電話株式会社通信ソフトウェア本部
Presently with Nippon Telegraph and Telephone Corporation

ルで実現した方が効率が良いことが知られている[6]．ユーザレベルでの実現例として，SunOS の Light Weight Process (LWP) ライブラリ[10]，フロリダ州立大の SPARC 用ライブラリ[4]等が存在するが，これらは，特定のアーキテクチャに依存しており，他のアーキテクチャでは動作しない．また，文献 2) に移植性の高いスレッドライブラリの実現法が示されているが，CPU の横取り（プリエンプション）機能の実現法は明らかでない．

そこで本研究では，（プリエンプションのある）軽量プロセス機構を実現するための移植性の良いライブラリの実現法を検討し，実際に作成した．このライブラリを **PTL** (Portable Thread Library) と呼ぶ．PTL は BSD スタイルのシグナル機構を備えた UNIX 上に実装可能であり，ほとんどの場合，make コマンドを実行するだけでインストールできる．ユーザはこれを利用することにより，軽量プロセスを利用したプログラムを様々なアーキテクチャ上で動作させることが可能となる．

アーキテクチャに依存しない軽量プロセス機構を実現する場合，スタックポインタの設定やコンテキストスイッチの方法が問題となる．本研究では，これらの問題に対する解決を行った．また，各スレッドが消費するスタックの量は一般に予測できない．このため，必要に応じて各スレッドのスタックが拡張できることが望ましいが，UNIX 上の軽量プロセスの実装では，スレッドのスタックは固定長であるものしか知られていない．PTL では，アーキテクチャによっては各スレッドのスタックを自動的に拡張することを可能にしている．

本論文では，2 章で PTL の設計方針について述べ，3 章で PTL の実現にあたって生じる問題と，その解決法を述べる．4 章では他の軽量プロセス機構と比較し，PTL の評価を行う．

2. 軽量プロセス機構に対する要求と方針

PTL に対する要求と，それに対する実現方針の概略を述べる．

- ライブラリ自身の移植性—ライブラリは特定の CPU に依存せず，なるべく多くのアーキテクチャで動作することが望ましい．このため PTL は特定の CPU に依存するコーディングを避け，すべてを C言語で記述することとする．また，アーキテクチャに依存する項目（スタックの成長方向，共有メモリの使用の可否，mprotect システムコールの使用の可否，シグナルハンドラを実行するのに必要なスタックのサイズ，等）は，PTL のインストール時に自動的に調査し，適切なシステムが構築できるようにする．また，動作対象としては，研究機関を中心に広く普及しているバークレー版の UNIX（正確には，BSD スタイルのシグナル処理が可能な UNIX）を対象とする．
- インタフェースの互換性—アプリケーションインタフェースは標準的であることが望ましい．このために，POSIX のスレッド拡張に対する規格案である P 1003.4 a/D 6（以下 Pthread と呼ぶ）[9] に準拠したインタフェースを提供する．
- 既存のライブラリ関数との親和性—printf 等の標準的なライブラリ関数は，シングルスレッドで動作することを前提としているため，マルチスレッドでは問題が発生する場合がある（FILE 構造体の更新が正しく行われない可能性がある等）．このため，PTL では標準的なライブラリ関数で問題が発生する可能性のあるものについては，相互排除を行うか，あるいはマルチスレッド対応の関数を新たに用意し，マルチスレッド環境でも実用上問題なく使用できるようにすることとした．

複数のプロセッサを備えた計算機では，同時に走行可能な複数のスレッドにそれぞれプロセッサを割り当て，真に並行に動作させたいという要求が存在する．しかし通常の UNIX では，プロセスに与えられたプロセッサは一つだけであるため，UNIX 上で動作することを目標としている PTL でもこの制限を受ける．このため，PTL では単一の CPU を用いて，時分割で複数のスレッドを実行することにする．

3. 軽量プロセス機構の実現上の留意点と対処法

軽量プロセス機構を UNIX の下で実現する際の留意点と，その対処法について述べる．

3.1 コンテキストスイッチ処理

スレッド間で制御の移動（コンテキストスイッチ）を行うためには，スレッドのコンテキストを保存，復元しなければならない．コンテキストには，CPU のレジスタの値などの，アーキテクチャ依存の情報が含まれる．これは，移植性を阻害する一因となる．

PTL では，コンテキストの保存，復元は文献 1) の方法で行う．これは，_setjmp, _longjmp 関数を用

いることによってコンテキストを保存，復元する方法であり，移植性のある唯一の方法と考えられる．

プリエンプションは，setitimer システムコールを用いて定期的にインターバル割り込み（SIGALRM）をかけ，SIGALRM ハンドラから PTL 内部のスケジューラを呼び出すことによって行っている．

スレッド1からスレッド2へプリエンプションが発生する場合，以下のようにしてコンテキストスイッチを行う（**図 1**）．

- SIGALRMが発生すると，UNIXカーネルは実行中のスレッド（ここではスレッド1）のコンテキスト情報をスタックに積む（図中の **UNIX Kernel Used**）．その後，（スレッド1の）SIGALRM シグナルハンドラの実行を開始する．
- ハンドラ中で _setjmp を行い，（スレッド1の）SIGALRM シグナルハンドラのコンテキストを保存し，将来の再実行に備える．その後，_longjmp を行い，以前に _setjmp で保存されていたスレッド2の SIGALRM シグナルハンドラのコンテキストを復元する．これによって，スレッド2の SIGALRM ハンドラの実行が再開される．
- スレッド2の SIGALRM ハンドラから復帰することにより，**UNIX Kernel Used** 中に保存されているコンテキスト情報を用いて，スレッド2の実行を再開する．

3.2 スタック処理

通常のプロセスのスタックは UNIX カーネルが管理しているが，スレッドのスタックはカーネルの管理外であるため，各スレッドのスタックの割り当て，拡張，スタックポインタの設定などはライブラリ側で行う必要がある．ここでは，これらに対する対処法を述べる．

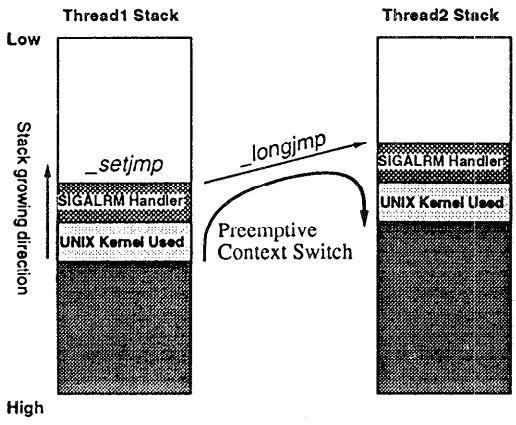


図 1 プリエンプションの流れ
Fig. 1 Flow of preemptive context switch.

本稿中では，スタックは高位アドレスから低位アドレスに伸びるものとする．以下，スタックの最上位アドレスを **Stack bottom**，最下位アドレスを **Stack limit** と呼ぶ．

3.2.1 スタックポインタの設定方法

スレッドの実行を開始する際には，スタックポインタを，開始するスレッドのスタックの **Stack bottom** に設定しなければならない．通常，このためにはアセンブラを用いて直接スタックポインタの値を設定するが，この方法では移植性を損なう．このため PTL では，この処理を BSD UNIX のシグナルスタック機能*を用いて，以下のように行う．

1. 開始するスレッドのスタック領域を sigstack システムコール（4.4 BSD では sigaltstack システムコールに変更されている）を使ってシグナルスタックとして使用することを宣言する．
2. 自分自身にシグナルを送ってシグナルハンドラに制御を移す．この時，シグナルハンドラはシグナルスタック（すなわちスレッドのスタック領域）を用いて実行される．
3. シグナルハンドラからスレッドのスタート関数を呼び出す．

この方法により，C言語とシステムコールだけでスタックポインタの設定が可能となった．

3.2.2 スタックのあふれ検出

各スレッドにはスレッドの作成時にプログラム中で指定したサイズのスタックを割り当てる．一番簡単で，移植性のあるスタックの割り当て方法は，ヒープ領域から割り当てること（**ヒープメモリスタック**）であるが，この方法では，スタックがあふれた際にプロセスが異常動作してしまう．

そこで PTL では，スタックのあふれ検出が可能な以下の2種類のスタックも提供している．これらのスタックでは，**Stack limit** より下位アドレスをアクセスするとシグナルが発生することを利用し，あふれ検出を行う．ユーザは，それぞれの特徴とコスト（4.2 参照）に応じてスタックを選択できる．

レッド・ゾーンスタック mprotect システムコールがサポートされている場合（SunOS 4, Ultrix 4.3, OSF/1 等）スタックをヒープから確保し，

* シグナルハンドラを実行する際に使用するスタックとして，あらかじめ指定したメモリ領域を使用することができる機能．

Stack limit から下位アドレスの一定領域を読み書き禁止に設定することで，スタックのあふれを検出できる☆.

共有メモリスタック 共有メモリ機構がサポートされている場合（SunOS 4, Ultrix 4.3, OSF/1, NEWS-OS 4, Mach 2.5 for Luna 88k 等），共有メモリセグメントをプロセスの仮想記憶空間に張り付け，その上にスタックを確保する．SunOS 4 等，共有メモリを張り付けるアドレスを比較的任意に選べる OS の場合，次に述べるようにスタックを自動的に拡張することも可能である．

3.2.3 スタックの自動拡張

一つのスタックは，仮想アドレス空間上で連続していなければならない．また，スタックには，それ自身の仮想アドレスに依存した内容が含まれている可能性があるため，使用中のスタックを別の仮想アドレスに移動することは不可能である．このため，スレッドのスタックがあふれた際，あふれから回復するためにはスタックを移動させずに，**Stack limit** の下位アドレス空間にメモリを割り当てることによってスタックを拡張しなければならない．

ヒープメモリスタックや，レッド・ゾーンスタックでは，スタックの下位アドレス空間には別のデータが格納されているため，スタックを拡張することは不可能である．一方，共有メモリスタックの下位アドレス空間には通常，メモリが割り付けられていない．共有メモリスタックがあふれた際に，この部分にメモリを張り付けることができ，かつスタックがあふれた際に実行しようとしていた命令を再実行することができれば，スタックのあふれから回復できる．PTL では，以下のようにして共有メモリスタックのあふれから回復を試みる．

1. PTL を初期化する際に，セグメンテーション違反によってシグナル（SIGSEGV）が発生したときに起動するシグナルハンドラを登録しておく．（SIGSEGV シグナルハンドラはシグナルスタックで実行されるように宣言しておく．ユーザスタックはあふれている可能性があるので，ユーザスタックの上でシグナルハンドラを動かすわけにはいかない．）
2. スレッドが与えられた共有メモリスタックを使い果たして，メモリの割り付けられていない領域をアクセスすると，SIGSEGV シグナルが発生し，SIGSEGV シグナルハンドラへ制御が移る．
3. SIGSEGV シグナルは他の原因によっても発生するため，シグナルハンドラでは，受け取ったシグナルがスタックオーバフローによるものかをチェックしなければならない．PTL では，セグメンテーション違反が発生したアドレスが，現在実行中のスレッドのスタックの **Stack limit** を越えて一定範囲に収まっていれば，受け取った SIGSEGV シグナルはスタックオーバフローによるものと見なしている．スタックオーバフローと判断した場合，シグナルハンドラでは，より大きな共有メモリセグメントへあふれたスタックの内容をすべてコピーしてから，**Stack bottom** のアドレスが変化しないように，もとの共有メモリセグメントと置き換える．（新たな共有メモリセグメントを，オーバフローを起こしたスタックの延長方向（**Stack limit** の下位アドレス）に単純に張り付けないのは，共有メモリセグメント数を節約するためである．）このようにしてシグナルハンドラから復帰すれば，スタックあふれを起こした命令から再開できる（**図 2**）．

スタックの自動拡張を行うためには，以下の条件が必要である．

（**条件 1**） 共有メモリを張り付けるアドレスの境界（アラインメント）が比較的小さいこと（ページサイズ境界等の緩やかな条件が望ましい）：例えば SunOS 4 (SPARC) では共有メモリセグメントを 4096 バイト境界で張り付けることができるが，NEWS-OS (R 4000) では 2M バイト境界でしか張り付けられない．共有メモリスタックはこの境界を単位として拡張せざるを得ないが，あまり巨大な共有メモリセグメントを確保することは無駄である．PTL では，この境界が 64K バイトより大きければ，スタックの自動拡張を行わない．

（**条件 2**） SIGSEGV シグナルハンドラからセグメンテーション違反を起こしたアドレスを取得できること：これは SIGSEGV シグナルが，スタックあふれに起因するものかどうかをチェックするために必要である．このアドレスの標準的な取得方法は存在しない．例えば，SunOS 4 ではシグナルハンドラへの第 4 引数として渡され，NEWS-OS, Ultrix 4.3 (R 3000) 等ではシグナルハンドラの第 3 引数が指す sigcontext 構造体の sc_badvaddr メンバに格納されている．

☆ この方法は文献 4) 等でも使われている．

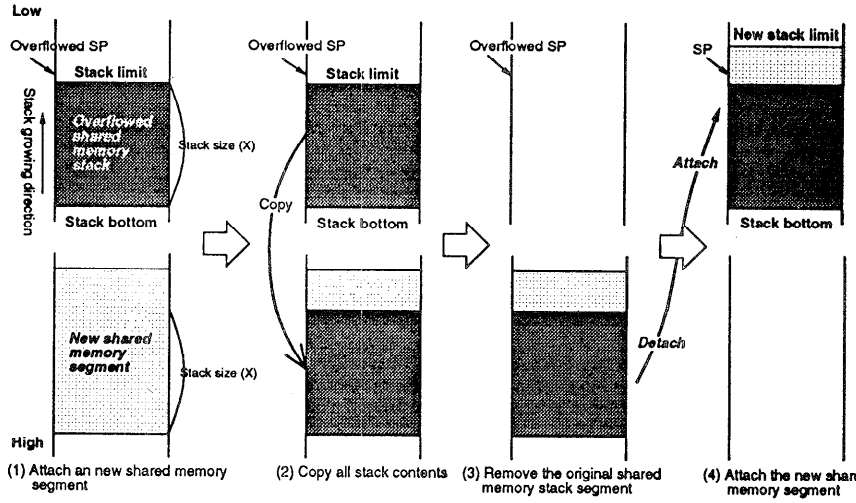


図 2 共有メモリスタックの拡張
Fig. 2 Expanding shared memory stack.

(**条件 3**) シグナルハンドラから，シグナルが発生する直前のスタックポインタの値を取得できること：これはスレッドのシグナルハンドラを実行する際のスタックポインタの値を決定するために必要である．伝統的な BSD UNIX では，この値はシグナルハンドラへ渡される第3引数の指す sigcontext 構造体の sc_sp メンバに格納されている．PTL では，sigcontext 構造体に sc_sp メンバがあるかどうかをチェックしている．

いくつかのアーキテクチャについて，これらの条件の適合性を示す (**表 1**)．すべての条件を満たす UNIX としては，例えば SunOS4 (SPARC および 68030)，OSF/1 などがある．

シグナルに対する対処 PTL では，プロセスに配送されたシグナルをスレッドに伝えるために，捕捉可能なすべてのシグナルに対して，汎用のシグナルハンドラを用意している．UNIX では，プロセスにシグナルが配送される際に，シグナルハンドラを実行するためのスタックの残りサイズがシグナル処理に必要な大きさを下回っていると，以下のような不都合が発生する．

- プロセスが異常終了する☆．(共有メモリスタックやレッド・ゾーンスタック等，Stack limit より下位アドレスのメモリにアクセスできない場合)．
- スタックの下位アドレスにあるメモリ領域を破壊する (ヒープ領域からスタックを確保した場合)．

☆ 捕捉できない無効命令シグナル (SIGILL) が送られる．

表 1 各 UNIX の共有メモリ機構の特性
Table 1 Characteristic of UNIX shared memory.

アーキテクチャ	条件1	条件2	条件3
SunOS4(SPARC)	4KB	○	○
SunOS4(68030)	16KB	○	○
Ultrix4.3(R2000)	4MB	○	×
OSF/1 V2.0(Alpha)	8KB	○	○
NEWS-OS4.1R(R4000)	2MB	○	×
LUNA UniOS-B(68030)	×	×	○

スレッドのスタックを自動拡張するためには，この問題に対処しなければならない．これに対する解決法を以下に述べる．

シグナルはいつ到着するか予測できないため，シグナルハンドラは十分な大きさを持つスタックで実行される必要がある．このため，シグナルハンドラは，あふれる可能性のあるスレッドのスタックではなく，十分な大きさを持つ別のスタック上で実行されるようにする (シグナルスタック機構を用いる)．

シグナルの到着の際にカーネルによってシグナルスタックに積まれる情報 (コンテキスト情報) は，割り込まれたスレッドを再開するために必要である．シグナルによってコンテキストスイッチが発生し，他のスレッドがスケジュールされる場合があるが，シグナルはさらにこのスレッドにも割り込む可能性がある．このため，スタック上のコンテキスト情報は，最大，スレッドの数だけ保持しなければならない．これを実現

するために，PTL ではスレッドごとにシグナルスタックを用意している（これを**スレッドシグナルスタック**と呼ぶ）．コンテキストスイッチの際には，プロセスレベルのシグナルハンドラがスレッドシグナルスタックを用いて実行されるようにシグナルスタックを切り替える．

この方法によって，シグナルによって割り込まれても安全にスレッドのスタックを拡張できる．

一方，スレッドスタックを拡張できないアーキテクチャ*では，シグナルハンドラは，通常のスレッドスタックの上で実行される（この場合，シグナル到着時にスタックが不足して異常終了する可能性はある）．

表 2 パフォーマンス
Table 2 Performance table.

測定項目	Sun 4 IPX		DEC 3000 AXP	
	PTL	FSU	PTL	OSF1
スレッドの生成（コンテキストスイッチの時間を含まない）				
共有メモリスタック	1320(1200)	—	496(270)	—
レッドゾーンスタック	1570(1010)	1600	783(543)	—
ヒープメモリスタック	1270 (770)	—	639(404)	971
（スタックキャッシュがヒットした場合）	170 (160)	330	66 (66)	?
スレッドの開始と終了	800	715	850	355
mutex のロック／アンロック（競合無し）	4.6	1	2.8	2.3
コンテキストスイッチ（yield）	147 (50)	37	70 (20)	35

単位：マイクロ秒
- Sun 4 IPX での値はスレッドスタックの自動拡張を許した場合．かっこ内は自動拡張を禁止した場合．
- Sun 4 IPX では，GNU malloc ライブラリをリンクしてある（PTL）．
- C コンパイラは gcc version 2 で，コンパイルスイッチとして -O2 を指定．
- OSF/1 のスレッドの生成時間のデータはヒープメモリスタックの欄に記載したが，実際にはどのようなスタックを用いているのかは不明である．

4. 評　価

ここでは，PTL の移植性，実行速度について述べる．

4.1 移植性について

本ライブラリは，現在以下のアーキテクチャで動作することを確認している．

SunOS 4, DEC Ultrix 4.3, DEC OSF/1 V 2.0, SONY NEWS 4, OMRON LUNA UniOS-B, OMRON LUNA-88K Mach 2.5, HP-UX, BSDI BSD/386, 4.4 BSD (DECstation)

BSD スタイルのシグナル機構を利用できる UNIX では，PTL は make コマンドを実行するだけでインストールできる**．

4.2 実行速度について

作成したライブラリの実行速度を測定するため，PTL と，アーキテクチャに依存した以下の二つの軽量プロセス機構を比較した（**表 2**）．これらはすべて Pthread に準拠している．

- フロリダ州立大の SPARC 用ライブラリ[4]（以下 FSU と略）(Sun 4 IPX, SunOS 4.1.1, SPARC)
- DEC OSF/1 の軽量プロセス機構（以下 OSFLWP と略）(DEC 3000 AXP, DEC OSF/1 V 2.0, Alpha)

PTL のスレッドスタックの自動拡張のコストを測定するため，自動拡張を禁止した場合も測定した．また，OSFLWP は Mach カーネルによってサポートされている．その他の機構はすべてユーザ空間内の実装である．

以下，測定した各項目について述べる．

- スレッドの生成（コンテキストスイッチの時間を含まない）

  PTL でスレッドスタックの自動拡張を行う場合，スレッドシグナルスタックのための領域を確保しなければならないため，余分な時間がかかっている．

  スレッドの生成時間のほとんどはスタックの確保（共有メモリセグメントやヒープ領域の確保など）にかかる時間であるため，この時間を節約できれば，高速にスレッドを生成できる．このため，PTL と FSU では不要になったスタックを回収して以降のスレッドの生成に備える機構（スタックキャッシュ）がある．スタックキャッシュを利用できた場合はかなり高速にスレッドを生成できることがわかる．OSFLWP では，スタックがキャッシュされるかどうかが不明である．

- スレッドの開始と終了

  全く処理を行わないスレッドを実行させることによって，スレッドの開始と終了にかかる時間の合計を測定した．FSU, OSFLWP に比べ，PTL の

* 共有メモリ機構がサポートされていない場合等．
** ヘッダファイルの違い等で若干の修正を要する場合がある．

方が遅くなっている．これは，PTL のスレッドの開始の処理にシステムコールを使ってシグナルを送っているためである（3.2.1 参照）．

- Mutex のロック，アンロック
  Mutex とは Pthread で規定されている相互排除のためのプリミティブである．Mutex のロック，アンロックは頻繁に用いられるため，高速に実行させることが望ましい．PTL はこの処理を int 型の大域変数を不可分にアクセスできることを用いて実現している．一方 FSU や OSFLWP では，この処理に（アーキテクチャに依存する）Test and Set 命令等を用いて高速化を図っているため，PTL の方が若干遅くなっている．
- コンテキストスイッチ
  スレッドスタックの自動拡張を行う場合，コンテキストスイッチの度にシグナルスタックを切り替える必要がある．これには3回のシステムコール*を要する．Sun 4 でスタックの自動拡張を行う場合にかなり時間がかかっているのはこれらのオーバヘッドのためである．また，スタックの自動拡張をしない場合に，DEC 3000 AXP で PTL の方が OSFLWP より高速にコンテキストスイッチを行っているのは，OSFLWP はコンテキストスイッチの度に Mach のシステムコールを実行しているためと予想される．

表2から，スタックの自動拡張にはいくらかコストを要することがわかる．このため高速性を追求する場合は，スタックの自動拡張機能はデバッグ時にのみ用いると良いと思われる．デバッグ時にスタックの自動拡張機能を用いてスレッドのスタック使用量を測定し，スレッド生成時に与えるスタックのサイズを予め十分な値に修正する．デバッグ終了後は，スタックの自動拡張機能を外して実行すれば，高速にコンテキストスイッチを行うことができる．

全体的には，アーキテクチャに依存した FSU 版のライブラリに比べて，遜色ないパフォーマンスが得られている．また，カーネルサポートの OSFLWP と比較すると，コンテキストスイッチ，スレッドの生成等ではユーザレベルの実装である PTL の方が速いことを確認した．

* sigsetmask システムコール×2，sigstack システムコール×1．

5. あとがき

本稿では，移植性の良い軽量プロセス機構の実現上の問題と，その解決法について述べた．

現在 PTL の β版を広域ネットワーク上で公開しており（ftp. center.osaka-u.ac.jp : /PTL/），既に多くのユーザによって利用されている．スタックポインタの設定処理を 10 行程度のアセンブリ言語で記述することにより，一部の SVR 4 UNIX マシンでも動作するようになっている．（動作確認は NEC EWS-4800/330 EWS-UX（R 4000），NCR System 3000（i 386）で行っている．）

我々の研究室においても，PTL を用いるオブジェクトコードを生成する LOTOS（分散システムや通信システムの仕様記述言語）コンパイラを作成しており，数千個のスレッドを並行に実行させるプログラムが稼働している[15]．

また，PTL を利用した広域分散型のマルチメディアシステムの構築も進めている[16]．このようなシステムでは，通常，各メディアごと，もしくは，メディア集合ごとにスレッドを割り当てるが，メディアの品質（QoS : Quality of Service）の制御を行おうとしたとき，各スレッドに対する CPU 資源の配分を自由に制御できることが望まれる．PTL を拡張して，このような制御機構を実現することは興味深い課題である．

デバッグ環境の改善も今後の重要な課題である．dbx や gdb 等の通常のデバッガを用いた場合，スレッドごとのデバッグができない，コンテキストスイッチを把握できない等の問題がある．そこで，マルチスレッドに対応した対話型のデバッガが望まれる．また，カーネルでスレッドをサポートしている OS（Mach, OSF/1 等）の上では，カーネルのスレッド機能を効率良く利用するような実現等も興味深い．

謝辞 PTL の SVR 4 への移植に御尽力頂いた，曽田哲之（SRA），堀井貞義（国際電気(株)）の両氏に深謝いたします．その他，貴重な御意見を頂いた PTL メーリングリスト（PTL@ics.es.osaka-u.ac.jp）の皆様に感謝いたします．

参考文献

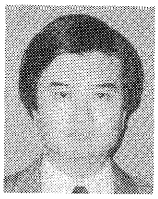
1) 多田好克，寺田 実：移植性・拡張性に優れたCのコルーチンライブラリーの実現法，信学論（D-I），Vol. J-73-D-I, No. 12, pp. 961-970 (1990).
2) 多田好克：機種に依存しない利用者 threads ライブラリ，情報処理学会研究会資料，92-PRG-8-

22, pp. 171-178 (1992. 8).
3) Mueller, F.: Implementing POSIX Threads under UNIX, *Proc. of Second Software Engeneering Research Forum*, pp. 253-261 (1992).
4) Mueller, F.: A Library Implementation of POSIX Threads under UNIX, 1993 Winter USENIX.
5) Cooper, E. C. and Draves, R. P.: C Threads, Department of Computer Science Carnegie Mellon University (1987).
6) 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).
7) Digital Equipment Corporation: DECthreads —Guide to DECthreads (1991).
8) Armand, F., Herrmann, F., Lipkis, J. and Rozier, M.: Multi-threaded Processes in CHORUS/MIX, *Proceedings of EEUG Spring '90 Conference*, pp. 1-13 (1990).
9) IEEE: Threads Extension for Portable Operating Systems (Draft 6), February 1992, P1003. 4a/D6 (1992).
10) Sun Microsystems: *System Services Overview*, pp. 71-106 (1988).
11) Stein, D. and Shah, D.: Implementing Lightweight Threads, *Proc. of the USENIX Conf.*, pp. 1-10 (1992).
12) Leffer, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S.: The Design and Implementation of the 4. 3 BSD UNIX Operating System, p. 208, Addison-Wesley Publishing Company, Inc. (1988).
13) 安倍広多, 松浦敏雄: 移植性に優れた軽量プロセスライブラリの実装法, *Proc. of the 21st jus UNIX Symposium*, pp. 78-89 (1993. 7).
14) 安倍広多, 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, 情報処理学会システムソフトウェアとオペレーティングシステム研究会資料, 62-10, pp. 73-80 (1994. 1).
15) Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K.: PROSPEX: A Graphical LOTOS Simulator for Protocol Specifications with N Nodes, *IEICE Trans. on Comm.*, Vol. E 75-B, No. 10, pp. 1015-1023 (1992).
16) 寺西裕一, 島村 栄, 松浦敏雄, 下條真司, 谷口健一: QoS を考慮したマルチメディアシナリオ記述言語, 情報処理学会情報メディア研究会資料, 18-1, pp. 1-8 (1994. 9).

(平成 6 年 4 月 14 日受付)
(平成 6 年 11 月 17 日採録)

安倍 広多 (正会員)

昭和 45 年生. 平成 4 年大阪大学基礎工学部情報工学科卒業. 平成 6 年同大大学院基礎工学研究科情報工学分野修了. 同年より, 日本電信電話株式会社通信ソフトウェア本部勤務. 工学修士. オペレーティングシステム, 並列分散処理等に興味を持つ.



松浦 敏雄 (正会員)

昭和 27 年生. 昭和 50 年大阪大学基礎工学部情報工学科卒業. 昭和 54 年同大大学院基礎工学研究科 (情報工学専攻) 博士後期課程退学後, 大阪大学基礎工学部情報工学科助手. 平成 4 年大阪大学情報処理教育センター助教授, 現在に至る. 工学博士. ユーザインタフェース, マルチメディア, 情報教育等に興味をもつ. ACM, IEEE, 電子情報通信学会等会員.

谷口 健一 (正会員)

昭和 17 年生. 昭和 40 年大阪大学工学部電子工学科卒業. 昭和 45 年同大学院基礎工学研究科博士課程修了. 工学博士. 同年同大学基礎工学部助手. 現在, 同情報工学科教授. 計算理論, ソフトウェアやハードウェアの仕様記述・実現・検証の代数的手法および支援システム, 関数型言語の処理系, 分散システムや通信プロトコルの設計・検証法などに関する研究に従事.