システムソフトウェアと
オペレーティング・システム 82－9
（1999．8．2）

UNIXプログラムをスレッドとして動作させる ProcThreadライブラリの設計と実現

安倍広多†, 松浦敏雄†, 安本慶一††, 東野輝夫†††

† 大阪市立大学 学術情報総合センター
†† 滋賀大学 経済学部
††† 大阪大学大学院 基礎工学研究科

あらまし

マルチスレッドは，複数の処理を同時に効率よく実行させるための機構である．スレッド間の通信やコンテクストスイッチは，プロセス間のそれと比較して高速であることが知られている．しかし，マルチスレッドには以下の問題がある．(1) スレッド間で行う相互排除や同期によって，スレッドの独立性が低くなるため，スレッドを別のプログラムで再利用することが難しい，(2) マルチスレッドプログラムを正しくコーディングするのは簡単ではない，(3) 既存の単一スレッド用のソースコードをマルチスレッドで利用するには適切な修正が必要．本稿では，マルチスレッドの高速性をなるべく保ちつつ，これらの問題を解決する一手法を提案する．提案する手法は，プロセスのアドレス空間内に複数の小さいプロセスを配置し，それぞれをスレッドとして動作させるという方法である．小さいプロセスは，別々の実行ファイルから，動的に読み込まれる．実際に，この方式をUNIX上に実装し，評価を行った．

キーワード　マルチスレッド，実行モデル，オペレーティングシステム，UNIX

ProcThread: A mechanism to execute UNIX program as a single thread

Kota ABE†, Toshio MATSUURA†, Keiichi YASUMOTO††, Teruo HIGASHINO†††

† Media Center, Osaka City University
†† Faculty of Economics, Shiga University
††† Graduate School of Engineering Science, Osaka University

Abstract

Multi-threading is an efficient mechanism to execute multiple jobs in parallel. Multi-threading are known to be faster than multi-processing because interprocess communication and interprocess context switching can be eliminated. However, multi-threading has problems; (1) Since each threads perform mutual exclusions and synchronizations, thread's independency are low and it is hard to reuse a thread in other programs. (2) Correct multi-threaded codes are difficult to program, (3) Traditional single threaded codes cannot be used without appropriate modifications for multi-threading. In this article, we propose a method to resolve these problems without losing performance of multi-threading. The method places multiple mini processes in an address space of a single process and executes each mini process using a single thread. Each mini processes are dynamically loaded from separate executable files. The method was implemented and evaluated on UNIX.

Keywords　Multi-Threading, Execution Model, Operating System, UNIX

1 はじめに

プログラミングの手法として，全体を一つのプログラムで構成するのではなく，いくつかのプログラムモジュールに分割して作成する方法が広く行われている．このようにして作成されたプログラムモジュールは，それぞれプロセスとして実行され，プロセス間通信を用いて互いに協調しながら動作する．しかし，プロセス間通信やコンテクストスイッチにはOSの介入が必要なため，実行速度が低下するという問題がある．これに対し，複数のプロセスを用いる代わりに，複数のスレッドを単一プロセスの中で用いる方法(マルチスレッド)がある．

マルチスレッドを用いると，プロセス間のコンテクストスイッチや，プロセス間通信を省略できるため，効率の良い実行が期待できる．しかし，マルチスレッドにも以下のような問題がある．

- マルチスレッドを用いたプログラムでは，複数のスレッドが同期や相互排除によって互いに密に結び付く．このため，あるスレッドが実行するコードを切り出して再利用することは，一般には容易ではない．

- マルチスレッド環境では，全てのスレッドが，あらゆる大域的なデータにアクセス可能である(アクセスを妨げるものは存在しない)．これは，非常に強力な機能であるが，反面，プログラマは大域的なデータに秩序を持ってアクセス(適切な排他制御を行う等)するように注意しなければならない．このため，マルチスレッドプログラムを正しく(バグがないように)コーディングすることは簡単ではない．

- 既存の単一スレッド用のソースコード(Legacyコードと呼ぶ)や，それから利用されるライブラリ関数は，スレッドセーフ(複数のスレッドから呼び出しても安全な関数)でない場合があるため，そのままではマルチスレッドプログラムで使用できない(流用が困難)場合が多い．また，それぞれのライブラリ関数がスレッドセーフかどうかは一般にOSによって異なるため，あるOSで問題がないプログラムでも，別のOSへ移植する場合に問題が出る可能性がある．

- 上記の理由で，既存の単一スレッドのプログラム(Legacyプログラムと呼ぶ)に，新たにスレッドを追加することも，容易ではない．

本稿では，マルチスレッドを用いることによる速度上の利点をなるべく保ちつつ，これらの問題を解決するプログラム開発方法およびその実行方式を提案する．

通常，マルチスレッド化されたプログラムでは，すべてのスレッドを構成するソースコードから，一つの実行ファイルを作成する．これに対し，本稿で提案する方式では，各スレッドを別々の実行ファイルとして作成した後，実行時にそれぞれの実行ファイルをプロセスのアドレス空間に動的に取り込んで，スレッドとして実行させる．この方式を，本稿ではProcThread方式と呼ぶ．この方式を実際にUNIX上に実装し，評価を行った．

2 ProcThread方式の特徴

マルチスレッドでは，単一の実行ファイルから生成されるプロセスイメージ中で複数のスレッドが動作するが，ProcThreadでは，単一プロセス中に，複数の実行ファイルから複数の小さなプロセス(μプロセスと呼ぶ)を生成し，それぞれを異なったスレッドで実行させる(図1)．本方式は，マルチプロセスと，マルチスレッドの中間に位置すると考えられる．通常のマルチスレッドよりもスレッド間の結び付きは低い．

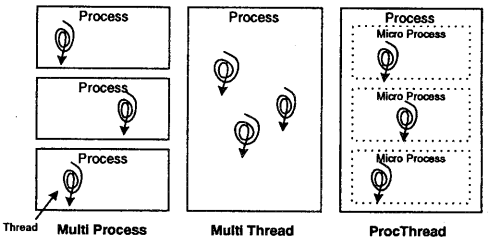

図1: 各方式の概念

本方式の特徴は以下の通りである(表1)．

- スレッドとファイルとの対応が明確となるため，スレッドの再利用が容易である．

- ProcThreadでは，μプロセス間で名前空間を共有しない(μプロセスAの大域変数aと，μプロセスBの大域変数aは異なる変数である)．プログラマはμプロセス間の通信は，意識して記述する必要がある．これによって，正式の手順を踏まずに大域データにアクセスしてしまうようなバグを減らすことができる．

- 複数のスレッドで同一のコードを共有することがないため，コードがスレッドセーフである必要がなくなる．このため，プログラマは呼び出すライブラリ関数がスレッドセーフかを意識する必要がなくなり，また，Legacy コードの流用が可能となる．
- μプロセスとしてプロセスに取り込むファイル(スレッド実行ファイルと呼ぶ) は，スレッドとして動作することも，通常の UNIX のプロセスとしても動作することが可能である．
- スレッド実行ファイルは，単一スレッドの実行ファイルと同じ手順，手間で作成できる．
- 単純なものならば，既存のプログラムをコンパイルし直すだけでμプロセスとして動作させることが可能である．

本方式の欠点は以下の通りである．

- 現在のところμプロセス間の通信に使用できるプリミティブが UNIX と同等のパイプ機構しかないため，μプロセス間の通信方式が限定される．(これは，μプロセス間プロシージャコールを実装して改善する予定である)．
- POSIX スレッド [1] 等を用いて，最初からマルチスレッドとして作成されたプログラムと比較すると，スレッドの生成に時間がかかる．また，メモリ使用量も多くなる (このため，ProcThread は，マルチスレッドを置き換えるものではない)．

表 1: マルチスレッドと ProcThread の比較

	マルチスレッド	ProcThread
実行のオーバヘッド	小	小
スレッド間の結び付き	密	粗
再利用性	×	○
バグの入り込みやすさ	大	小
既存のソースの流用	×	○
通信のしやすさ	○	△
メモリの使用量	小	大

関連研究としては，効率化のために，複数のプロセスを一つのアドレス空間に配置する機能を持つ OS が研究・開発されている [2, 3]．これらの OS のプロセスを，ProcThread のμプロセスと見なすと，両者の実行モデルは類似している．しかし，ProcThread では，広く普及している UNIX で，カーネルを変更することなく実装したところが特徴であり，また，ProcThread はユーザレベルスレッドとして実装しているため，コンテクストスイッチ等で OS を経由する必要がないという利点もある．

また，プロセス内に複数のスレッドを配置することは一般的であるが，本稿で提案するような，各スレッドが実行するコードを別々の実行ファイルから取得する手法は，提案されていない．

3 ProcThread 方式の詳細

ProcThread では，プロセスのアドレス空間内に，ProcThread のサービスを提供する ProcThread コアと，1 つ以上のμプロセスが配置される．それぞれのμプロセス毎に，それを実行するスレッドが存在する．このように，ProcThread では，UNIX プロセスのアドレス空間内に，小さな OS と，1 つ以上の小さなプロセスを配置するような構造になっている (図 2)．

ProcThread 方式を用いるプログラム P を実行するには，ProcThread を起動するプログラム (procthread) に，P のファイル名を与えて起動する．これによって，プロセスのアドレス空間に，ProcThread コアと P のμプロセスが配置され，実行される．以下，詳細を述べる．

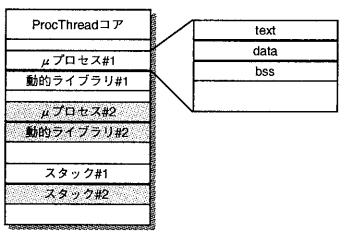

図 2: アドレス空間のレイアウト

3.1 ProcThread コア

ProcThread コアは，プロセスのアドレス空間中に，いわばプロセス内 OS として存在する．提供する主な機能は以下の通りである．

- スレッド管理(生成・消滅・スケジューリング等)
- ファイル記述子管理
- スレッド間通信機構

スレッドは，これらの機能に ProcThread システムコールを用いてアクセスする．ProcThread シス

テムコールは，プロシージャコールとして実装されるため，オーバヘッドは少ない．これら以外のサービスは，UNIX カーネルが提供する．スレッドが UNIX のシステムコールを発行すると，通常と同じ仕組みで UNIX のシステムコールが実行される (図 3)．

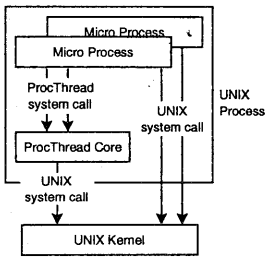

図 3: ProcThread のシステムコール

3.2 ProcThread の実行ファイル

スレッド実行ファイルは，基本的に動的リンクされた UNIX の実行ファイルであり，どのアドレスにロードされても動作するように，再配置情報を追加してある．

スレッド実行ファイルの作成は，通常のプログラムの作成と同じように，ソースファイルをコンパイル・リンクすることによって行う．異なるところは，再配置情報を付加する特別なリンカ (ld) を用いることと，特別なスタートアップルーチン (crt0.o) をリンクすることだけである．これを容易にするため，専用のコンパイルドライバを用意している．

スレッド実行ファイルの形式は，UNIX の実行ファイル形式と互換性があるため，シェルから実行したり，通常のデバッガを用いてデバッグすることも可能である (4.1.5 参照)．

3.3 ProcThread のスレッド

プログラマは，スレッド毎にスレッド実行ファイルを用意する．スレッドの実行は main 関数から始まると考えて良い．main 関数には，ANSI C の標準的な引数 (argc, argv, envp) が渡される．

複数の μ プロセスがプロセスに存在する場合，アドレス空間は共有するものの，それぞれが利用するアドレスの領域は独立している．リンクされるライブラリ (動的リンクされるものを含む) も μ プロセス毎に存在するため，使用するライブラリはスレッドセーフでなくても構わない．

環境変数は，μ プロセス毎に固有であり，μ プロセスを作成したスレッドから引き継がれる．

3.4 μ プロセスの生成と消滅

μ プロセスの生成 (スレッドの生成) は，以下の関数を用いて，スレッド実行ファイル (path) と，スレッドに渡す引数 (argv)，環境変数 (environ)，ファイル記述子マップ (fdmap, 3.5 参照)，属性 (flag) を指定して行う．

```
procthread_create(int *id, char *path,
    char **argv, char **environ,
    struct fdmap *fdmap, int flag);
```

生成に成功すると，id に生成したスレッドの識別子が返る．これによって，スレッド実行ファイルがプロセスのアドレス空間に読み込まれ，そのスタートアップルーチンからスレッドの実行が開始される．

同一のスレッド実行ファイルを用いてスレッドを複数回生成する場合，flag で指定することにより，メモリ上の μ プロセスを再利用し，2 回目以降のスレッドの生成を高速化できる (詳細は 4.1.3)．UNIX のプログラムでは，system 関数などを用いて，外部プログラムを呼び出すということがよく行われるが，ProcThread では，この再利用機能を用いることにより，外部プログラムをスレッド実行ファイル形式としておけば，2 回目以降はスレッドとして高速に実行することが可能である．

また，flag で指定することにより，指定したスレッド実行ファイルを通常の UNIX プロセスとして生成することも可能である．これは，スレッドのデバッグに有効である．

スレッドが exit 関数を呼び出すと，当該スレッドの実行は終了する．再利用が指定されていなければ μ プロセスも消滅する．Legacy プログラムでは，プロセス終了時に利用していた資源が消滅することを前提に，確保した資源 (オープンしたファイル記述子，mmap した領域等) を明示的に解放しない場合が多い．ProcThread の exit では，スレッドが確保したこれらの資源の解放も行う．これによって，Legacy コードの流用が容易になり，スレッドの終了時に資源の解放を忘れるというバグを防ぐことができる．

3.5 ファイル記述子管理

通常のマルチスレッドプログラムでは，ファイル記述子は複数のスレッドで共有するが，ProcThreadではμプロセス毎に独立したファイル記述子(スレッドファイル記述子と呼ぶ)を持つことができる(スレッド1のファイル記述子0と，スレッド2のファイル記述子0が別のものを指すことができる)．これは，ProcThreadコアが，UNIXカーネルが管理するプロセスのファイル記述子との間で変換を行うことによって実現する．ProcThreadのパイプ機構(3.6で後述)は，UNIXのパイプと同様，ファイル記述子を用いて，スレッド間の通信を行う(図4)．

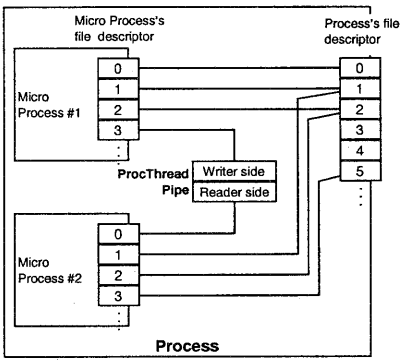

図4: スレッドファイル記述子

UNIXでは，fork後にdup2等を用いてファイル記述子をコピーしてからexecすることによって，入出力のリダイレクションを行うが，ProcThreadでは，スレッドの生成にfork&execモデルを使用していないため，リダイレクションは次の手順で行う．

まず，ProcThreadシステムコールでファイル記述子マップ(スレッドと無関係に存在するファイル記述子テーブル)を取得する．この上で，dup2やcloseに相当する操作を行い，生成するスレッドのファイル記述子テーブルを準備する．スレッドを生成するときに，このファイル記述子テーブルを与えることによって，スレッドの入出力のリダイレクションを行う．

3.6 スレッド間通信機構

現在のところ，ProcThreadではスレッド間通信機構として，UNIXのパイプ機構と同様の機構を用意している．他に，μプロセス間プロシージャコールも提供する予定であるが，まだ実装できていない．ここでは，パイプ機構についてだけ述べる．

ProcThreadのパイプ機構によって，スレッド間の通信をUNIXカーネルを介さず，高速に行うことができる．ProcThreadのパイプは，UNIXのパイプと同じインタフェースで動作する．

procthread_pipe(int fds[2]);

3.5で述べた方法で，スレッドの標準入力(ファイル記述子0)，標準出力(ファイル記述子1)をProcThreadのパイプに割り当てることにより，UNIXのフィルタコマンドをスレッド実行ファイル形式にするだけで，スレッドとして利用可能である．

4 実装

ProcThreadを，フリーUNIXとして公開されているNetBSD-1.4(i386)に実装した．

4.1 ProcThreadコア

ProcThreadコアは，我々が研究・開発している，移植性の高いマルチスレッド機構PTL[4]をベースに開発した．PTLは，ユーザレベルでのスレッド実装であり，POSIXスレッドのほぼ全ての機能を備えている．ProcThreadでも，PTLと同様，スレッドはユーザレベルスレッドとして実現している．以下，実装の概略を述べる．

4.1.1 スレッド実行ファイルの配置

スレッド実行ファイルの配置は，以下のように行う(図5)．

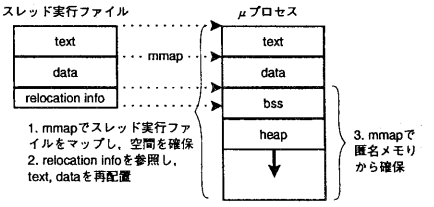

図5: μプロセスの生成

1. スレッド実行ファイルをmmapでマップする．brkやsbrkの呼び出しによってスレッドのbss領域が拡大できるように，実際よりも大きい領域を確保する．
2. ファイルの再配置情報を参照して，text, data領域の再配置を行う．

3. bss 領域は，mmap で匿名メモリ[1] をマップする．bss 領域の拡大のため，スレッド実行ファイルのヘッダで指定された bss 領域のサイズよりも大きく確保しておく[2]．

スレッドのスタックは，ProcThread コアが管理するメモリ領域 (mmap によって作られたセグメント) から割り当てる．

4.1.2 実行時リンク機能

通常の動的リンクされた UNIX プログラムは，実行時にスタートアップルーチンが実行時リンカ (ld.so) をメモリ上に mmap して，実行時リンク処理を行う．

ProcThread では，ProcThread コアに実行時リンク機能を内蔵した．これにより，mmap や実行時リンカの初期化処理等を省略でき，実行時リンク処理を効率化できる．内蔵する実行時リンカは，ほぼ ld.so そのものであるが，一部，実行ファイルが固定アドレスにロードされることを前提としているコードがあるため，その部分を修正した．

4.1.3 μプロセスの再利用

ProcThread のスレッド (μプロセス) の生成は，通常のスレッドの生成と比較すると高価な処理である (5.1 参照)．このため，3.4 で述べたように，スレッドの実行が終了した後でもメモリ上の μプロセスを保持しておき，後で同一のスレッド実行ファイルから μプロセスを生成する際に再利用するようにした．

μプロセス中の text 領域はスレッドの実行によって変更されることはないが，data, bss 領域は変更されるため，これらの領域は，再利用時に初期値に戻す必要がある．このため，実行時リンク処理が終了した時点での data 領域の内容を一時ファイルにコピーしておく．再利用時には，data 領域にこのファイルを mmap する (この時，メモリに書き込んでも，ファイル自体は書き変わらないモード (MAP_PRIVATE) を使用する)．bss 領域は，mmap によって新しいメモリセグメントを割り当てる．これにより，再利用時には高速に μプロセスを再生成できる．

[1] mmap で MAP_ANON を指定することによって得られるメモリセグメント．

[2] brk や sbrk の契機で新たなセグメントを mmap しなおしても良いはずだが，正常に動作しなかった．これは NetBSD の仮想記憶機構の問題と思われる．

4.1.4 スレッドの実行

UNIX カーネルがプログラムを実行 (exec) する場合，引数や環境変数をユーザのスタックに積んでから，プログラムのエントリポイントを呼ぶ．ProcThread でのスレッドの実行開始処理でも，全く同じようにスタックを設定し，スレッド実行ファイルのエントリポイントを呼ぶ．これによって，制御はスレッド実行ファイルにリンクされたスタートアップルーチン (4.2 参照) に移る．

なお，ここで設定しなければならないスタックの内容は，UNIX によって異なるため，この処理は UNIX 毎に変更する必要がある．

4.1.5 ProcThread システムコール

ProcThread システムコールを実現するために，スレッド実行ファイルには，専用の共有ライブラリ (libProcThread.so) を動的リンクさせる．この共有ライブラリに，ProcThread システムコールのスタブ関数が含まれる．スタブ関数は，ProcThread コアのシステムコールエントリ関数をプロシージャコールすることによって，ProcThread システムコールを実行する．(スレッドには，エントリ関数のアドレスは環境変数で渡す)．

スレッド実行ファイルは，2 通りの実行モード (ProcThread のスレッドとして動作するスレッドモードと，直接 UNIX の exec システムコールによって実行される UNIX プロセスモード) があり，モードによって以下のように動作を変更する．

スレッドモード　スレッドとして動作する場合，幾つかのシステムコールを UNIX カーネルではなく，ProcThread コアで実行する必要がある (例えば，スレッドの終了時に exit システムコールを発行してしまうと，プロセス全体が終了してしまう)．これを避けるために，libProcThread.so が，以下に挙げるシステムコールを ProcThread コアで横取りする (必要に応じて，ProcThread コアで UNIX システムコールを発行する)．

brk, sbrk
: スレッド毎に bss 領域が存在するため，bss 領域を拡大するこれらのシステムコールは ProcThread コアで横取りする．

exit
: exit は，スレッドを終了させるために用いる．

ファイル記述子を扱うシステムコール
スレッドファイル記述子を扱えるようにする.

chdir
カレントディレクトリは，スレッド毎に管理する.

プロセスの資源を確保・解放するシステムコール
スレッド終了時に，スレッドが解放し忘れた資源を解放するため，プロセスの資源を確保·解放するシステムコール (mmap, munmap, open, close 等) を横取りし，スレッドが確保した資源を常に記憶しておく.

UNIX プロセスモード　スレッド実行ファイルが UNIX プロセスとして動作する場合は，ProcThread コアが存在しないため，ProcThread システムコールをどのように解決するかが問題となる. この場合，ProcThread システムコール相当の処理を，ProcThread コアなしで行う互換ライブラリを動的リンクさせることによって，この問題を解決する (例えば，procthread_create() は，fork, dup2, execve 等で置き換えることができる). これによって，スレッド実行ファイルを全く変更せずに，UNIX プロセスとして動作させることができる.

4.1.6 パイプ機構

UNIX のパイプは，write されたデータをカーネル内に一旦コピーし，read 時に再度コピーすることによって実装されている (2 コピー). これに対し，ProcThread では，あるスレッド (W) がパイプに write した場合，他のスレッド (R) が read するまで，W は待たされる (逆に，先にスレッドが read を発行した場合，write するスレッドが現れるまで待たされる). read が発行されると，W は，書き込むデータを read で指定する領域にコピーする. このように，ProcThread では 1 コピーでデータの転送を行う. また，ProcThread のパイプの読み書きには UNIX のシステムコールが不要である. これらの理由で，ProcThread のパイプは高速に動作する.

4.2 スタートアップルーチン (crt0.o)

スレッドの実行が開始されると，まずスレッド実行ファイルにリンクされたスタートアップルーチンが実行される. このルーチンはスレッドとして起動されたのか，通常の UNIX プロセスとして起動されたのかどうかを環境変数で判定する.

前者の場合，ProcThread コアの実行時リンク機能を呼び出す. 後者の場合，通常のスタートアップルーチンの処理 (実行時リンカ (ld.so) のマップ等) を行う. いずれの場合でも，その後，スレッドの main 関数に制御を移す.

4.3 リンカ (ld)

スレッド実行ファイルを生成するために，リンカに手を加えた. UNIX の実行ファイル (a.out 形式) とオブジェクトファイル (.o ファイル) は同一のファイルフォーマットであるため，実行ファイルにも，再配置情報を格納できる構造になっている. これを利用して，実行ファイルにも再配置情報を残すようにした.

5 評価

作成した ProcThread 処理系の評価を行った. OS として NetBSD-1.4(i386)，ハードウェアとして IBM-PC 互換機 (Pentium 133MHz, Memory 64MB) を使用した.

5.1 スレッド生成・終了

何もしないプログラムをスレッド実行ファイルとして用意し，以下の時間を測定した.

- ProcThread でスレッドを生成，実行し，終了 (join) するまでの時間
- 上記と同じだが，μプロセスを再利用した場合
- UNIX で，fork&exec し，実行，終了 (wait) が完了するまでの時間
- (参考)PTL(POSIX thread) で何もしないスレッドの生成，実行，終了に要する時間

結果を表2 に示す. ProcThread で，fork&exec よりも時間がかかっている主な原因は，μプロセスの配置をするために，システムコール (open, mmap×2) と，再配置処理を行う必要があるためである. 再利用した場合には，これらのシステムコールと再配置，さらに実行時リンク処理まで省略できるため，PTL のスレッド生成時間とほぼ同じ時間で実行できている. このため，複数回，同一のμプロセスを生成・実行するコストは非常に低いと言える.

表 2: スレッド生成・終了の時間

	時間 (msec)
ProcThread	50
ProcThread(再利用)	1.20
UNIX fork&exec	21
POSIX thread	1.18

5.2 パイプ機構

ProcThread のパイプ機構の速度を測定するため，以下の実験を，ProcThread のパイプを用いた場合と，UNIX のパイプを用いた場合の両方で行い，かかった時間を測定した (表 3).

- ProcThread の 2 つのスレッド (あるいは UNIX の 2 つのプロセス) 間で，パイプを経由して 4K バイトのデータを 5000 回転送する.
- 実際のアプリケーションとして，以下のシェルパイプライン相当を実行する (ProcThread の場合，cat, grep コマンドはスレッド実行ファイルとしておく).

  `cat /usr/share/dict/words | grep zonation`
  (words は約 2.4MB のテキストファイル)

表 3: パイプによる転送速度 (msec)

	ProcThread	UNIX
4KB (5000 回)	980	2700
`cat | grep`	380	620

ProcThread のパイプは，UNIX のパイプよりも 3 倍程度高速であり，実際のアプリケーションでも高速化の効果が得られることがわかる.

5.3 Legacy コードの流用

NetBSD のいくつかのフィルタコマンドが，ソースに変更を加えることなくスレッドとして動作するかどうかを調査した．調査した範囲 (`grep`, `cat`, `sort`, `uniq`, `wc`) では，スレッド実行ファイルとしてコンパイルし直すだけで，すべてのコマンドがスレッドとして正常に動作することを確認した.

6 まとめ

本稿では，従来のマルチスレッドプログラミングでの問題点のいくつかを改善する ProcThread 方式を提案し，UNIX 上で実装，評価を行った．ProcThread は，スレッドの再利用性が向上する，バグが入り込みにくい，Legacy コードの流用が容易となる，という利点を備える．実験により，スレッドの生成はプロセスの生成 (fork&exec) より時間がかかるものの，スレッドの再利用機構によって，同一スレッドの 2 回目以降の生成は非常に高速に行えること，スレッド間通信機構であるパイプ機構が UNIX のそれよりも高速に動作すること，既存のプログラムを全く変更せずにスレッドとして動作させることが可能であること，UNIX のフィルタコマンドを μ プロセスとして取り込むことにより，シェルパイプラインを高速に実行できることを示した.

今後の課題は以下の通りである.

- 本文中でも触れたが，現在，スレッド間通信機構としては単純なパイプ機構しか提供していない．今後，μ プロセス間プロシージャコールを実装し，評価する予定である.
- 現在，ProcThread では μ プロセス毎に，スレッドは 1 つだけ存在するが，1 つの μ プロセス中で，POSIX スレッドのような通常のマルチスレッドプログラミングを行いたい場合も考えられる．これをサポートすると，通常のマルチスレッドは，ProcThread の特別な場合 (μ プロセスがプロセス中に 1 つしか存在しない) ということになる.

参考文献

[1] ISO/IEC: *Information technology - Portable Operating System Interface (POSIX) - Part1: System Application Program Interface (API) [C Language]*, IEEE (1996).

[2] 谷口秀夫, 長嶋直希, 田端利宏: 単一仮想記憶と多重仮想記憶を共存させたヘテロ仮想記憶の実現, 情処研報, Vol. 98, No. 33, pp. 87–94 (1998).

[3] 毛利公一, 大久保英嗣: マイクロカーネル Lavender の設計と開発, 電子情報通信学会論文誌, Vol. J82-D-I, No. 6, pp. 730–739 (1999).

[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).